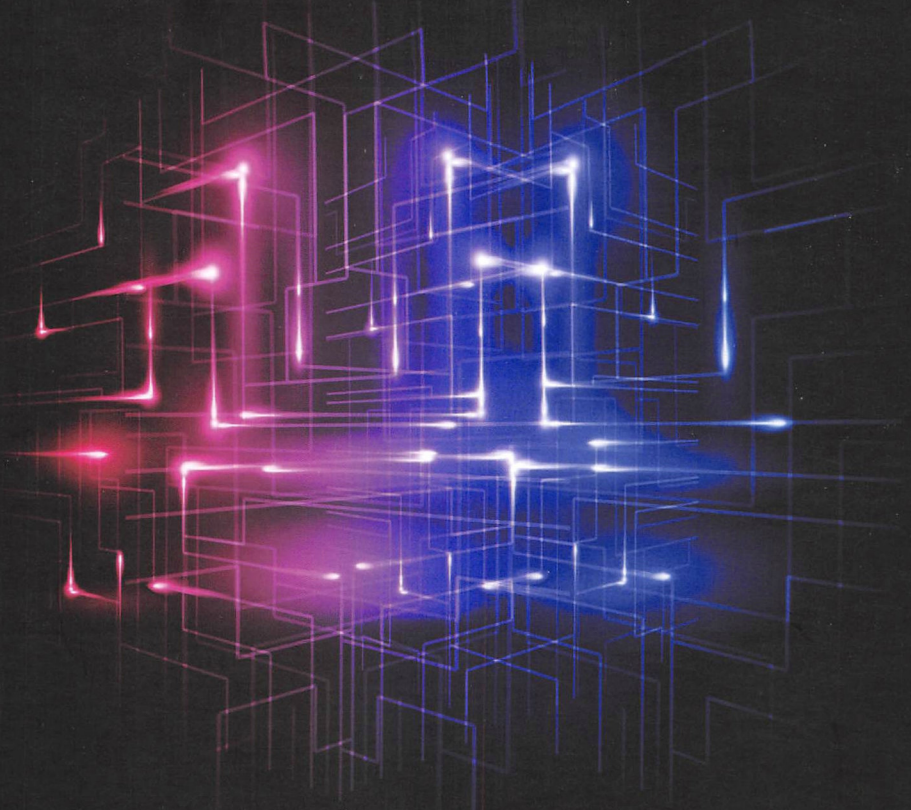
版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

仅供非商业用途或交流学习使用







Building High Availability and Distributed Applications by Akka

# Akka实战

## 快速构建高可用分布式应用

杜云飞◎ 著

资深Java技术专家在大量实践后所得的经验与教训的总结

以实战为导向，从Akka架构、组件、工具包、分布式、集群、微服务等多个角度展开，为构建高可用、高性能的分布式应用提供了详尽指导



机械工业出版社
China Machine Press



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



## 内容简介

Akka 是一款非常优秀的分布式并行应用框架，它为快速构建高可用、高性能的分布式应用提供了良好的解决方案。

本书是资深 Java 技术专家在大量实践后所得的经验与教训的总结，以实战为导向，从架构、组件、工具包、分布式、集群、微服务等多个角度对 Akka 进行了详细讲解，为快速构建高可用、高性能的分布式应用提供了详尽指导。每个知识点都配备了详细的代码示例和配置说明，实操性极强。

全书一共 10 章：

第 1 章 全面介绍了 Akka 的概念、应用场景和系统架构，为技术团队全面了解 Akka 和技术选型提供了参考依据。

第 2~5 章 详细讲解了 Akka 的各种模型 & 组件的原理及用法，如 Actor、线程调度（Dispatcher）、邮箱和消息路由等。

第6章 介绍了Akka的一些实用工具包，如Future、Event Bus、Akka Streams 等。

第 7~8 章 主要讲解了 Akka 分布式 & 集群相关的内容，如远程 Actor、远程路由、事件监听、集群搭建、集群分片、持久化等。

第 9 章 讲解了如何使用 Akka 构建 HTTP 和 WebSocket 服务，并实现了一个完整的图文聊天系统。

第 10 章 介绍了一款“全家桶式”的微服务框架 Lagom，它基于 Akka 和 Play 而构建，提供了一整套微服务解决方案。



仅供非商业用途或交流学习使用

华章IT
HZBOOKS | Information Technology

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用





仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



# Akka实战

## 快速构建高可用分布式应用

杜云飞◎ 著

Building High Availability and
Distributed Applications by Akka



机械工业出版社
China Machine Press



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



图书在版编目（CIP）数据

Akka 实战：快速构建高可用分布式应用 / 杜云飞著 . —北京：机械工业出版社，2018.4

ISBN 978-7-111-59685-1

I. A… II. 杜… III. JAVA 语言—程序设计 IV. TP312.8

中国版本图书馆 CIP 数据核字（2018）第 067315 号

# Akka 实战：快速构建高可用分布式应用

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：李 艺　　责任校对：李秋荣

印　　刷：北京市兆成印刷有限责任公司　　版　　次：2018 年 4 月第 1 版第 1 次印刷

开　　本：186mm × 240mm 1/16　　印　　张：13.25

书　　号：ISBN 978-7-111-59685-1　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



# Preface 前 言

## 为何写作本书

在大力提倡互联网+的今天，人们在PC、手机、Pad上安装了各式各样的软件服务或APP，享受着互联网技术带来的高效和便捷。同时，随着这些产品的高速发展，用户规模急剧扩大，系统承载能力会逐渐达到瓶颈，一些早期不常出现的问题也会慢慢浮出水面。实际上，系统开发者们也越来越重视软件的稳定性、高可用性、可扩展性对用户体验的影响，大家纷纷使出浑身解数，实践出多个构建大型应用所需要的技术架构。在这些架构方案中，有几个词始终不绝于耳，那就是："高并发""分布式""高可用"。

对于很多工程师来讲，这几个词可谓"既爱又恨"。"爱"是因为大家都希望能掌握大型分布式系统架构所涉及的各个知识点。这样，大家在实现产品不断迭代更新，并持续服务好用户的同时，也能够进一步提升自己的技术实力和技术视野，可谓"产品"和"技术"共同成长。"恨"是因为很多产品并没有机会"跑"成大型系统，工程师们也就没有机会去解决大型系统才会出现的可扩展性问题，再加上目前市面上的一些分布式架构解决方案确实都存在不小的门槛，让大家认为任何分布式系统（学习起来）都是如此复杂。从某种程度上，这些都限制了工程师的自我提升。后来，我有幸接触到了一些较大型的系统，也逐渐开始承担起技术选型的工作，以我的经验来看，当你对整个系统架构有一定决策权的时候，就必须考虑你所选的架构是否足够稳定、可控、可依赖，更直白点说，即是否足够"简单"。因为只有简单的东西，才是可控的、低风险的。所以在当时，我所有的技术选型都以"简单"作为出发点，不希望将事情做得更复杂。就在我们打算为某个业务产品实现一套实时数据处理系统时，我接触到了Akka框架，在经过初步的了解后，我们开始尝试搭建Demo工程。最初我们只需要一个足够简单的单机程序，不用考虑太多的可扩展性问题，而Akka在单机上的并发性能也足以支撑我们当时的业务需求。但随着产品不断迭代和更新，数据处理所需要的系统资源突破单机的瓶颈，这时就不得不对其做水平扩展，即分布式&集群。我们发现，使用Akka做这类扩展是如此简单，无论是单机还是分布式，它都提供了统一的编程模型，这正好符合我的"简单"原则。Akka并非大型项目才可以使用，只要场景合适，



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用





它可以从“小”用到“大”。所以当你想要学习一门简单够用的分布式并行框架时，可以从Akka开始。

Akka虽然好用，但是在国内的知名度并不高（特别是和Java界的那几座大山相比）。我个人认为，出现这样的情况，可能有两个原因：一是Akka是基于Scala开发的，虽然同为JVM上的语言，也提供了Java API，但是Scala还略显小众（语言本身还是挺棒的），大家接受度不高；二是Akka的中文资料非常匮乏，这让很多工程师学习起来比较吃力，并且在出了问题后，不太容易找到合适的解决方案。我曾有幸被好几家企业邀请做一些这方面的技术咨询和培训，期间和小伙伴儿们一起踩过不少坑，但是总算爬出来了（笑），并且还留下了一些自己做的笔记和资料，所以就想写一本Java版的Akka书籍，也算是为普及Akka框架做点小贡献。再后来，一个偶然的机会认识了机械工业出版社的杨福川老师，了解了书籍出版的相关事宜，在他的鼓励下，我开始走上写书的道路。

写书对我来讲，远比写代码要困难得多。写代码时，我只需要考虑当前业务场景下的功能实现，并且对于技术的运用更多靠的是一种习惯——觉得该这么写，不用考虑“why”，也不用考虑具体的细节。但写书需要更加全面地了解整个框架结构，将零散的知识“点”串成一条“线”，然后连成一个个知识“面”。虽然以前我有记笔记和写博客的习惯，但是要把这些整合成一本书，还是花费了不少精力。另外，写作时间对我来说也是个问题。在决定要写这本书时，我正处于独立创业中，并且刚从上一个项目中抽出身来，时间相对来说比较充裕，所以趁热打铁，很快交出了第一稿。正当我为自己的写作进度感到开心时，一个新的创业机会向我招手了，然后我来到了现在这家公司。加入新公司后，我几乎把所有精力都投入到新产品的研发和管理工作上，所以写书一事也只能暂缓了。在这期间，我一直得到家人、朋友，包括福川老师的支持和鼓励，我对完成本书的愿望也越来越强，所以不敢（也不想）有任何懈怠，一直尽可能挤出空闲时间用于写作。终于，在搭上了多个“不那么忙”的周日后，我完成了本书。

## 本书特色

据我了解，目前国内还没有一本真正意义上的Akka原创书籍，而网上流传的相关资料同质化又比较严重，同时对Java版进行全面介绍的也非常少，这些都对初学者产生了很大干扰。本书以Java版为基础，对Akka的各个知识点都做了较为全面的介绍，并且每个知识点都附带了详细的代码示例，也对配置进行了说明，相信读者朋友们可以很快上手。

## 本书读者对象

- Java工程师
- 系统架构师
- 分布式系统爱好者
- 高校师生及IT培训机构相关人员



仅供非商业用途或交流学习使用

## 如何阅读本书

本书一共分为 10 章，全面介绍了 Akka 的基础架构、Actor 模型、常用组件、分布式 & 集群、微服务等。下面是各个章节的简介及阅读说明。

第 1 章主要介绍了 Akka 的技术背景、应用场景及架构体系，为大家日后做技术选型提供参考依据。

第 2 章全面介绍了 Actor 的相关概念、常见用法、生命周期、容错处理等。Actor 可谓 Akka 最核心的组件，所有其他功能都是围绕它来构建的，所以对其理解越深刻，越有助于后续内容的学习。

第 3 章介绍了 Dispatcher 的用途和常见配置方法。对其进行合理配置，可以有效提高系统吞吐率。

第 4 章介绍了邮箱的常见类型及配置，同时给出了一个自定义邮箱类型的简单案例。

第 5 章介绍了消息路由器的基本概念和创建方式（Pool 和 Group），然后分别演示了“广播”“最快响应”等常见路由策略。虽然本章的示例都是基于本地系统的，但实际上它可以无缝切换成远程模式，即在分布式集群环境下（具体会在第 7 章和第 8 章进行介绍），路由功能也是完全可用的。

第 6 章介绍了 Akka 的一些实用工具包，比较重要的有 Future、Event Bus（事件总线）、Akka Streams 等。其中 Future 可以接收 Actor 的计算结果，并通过多种方式对结果进行处理；事件总线实现了基于发布 - 订阅的消息流处理，解决了耦合问题；Akka Streams 是基于 Reactive Streams 的一个实现，它提供的“背压”（Back Pressure）机制用于解决大规模数据处理带来的生产和消费速度不均的问题，并以此实现流控。Akka Streams 的内容较多且复杂，大家可以通过本章入门，更多详细的内容请参考官方文档。

第 7 章和第 8 章主要介绍了 Akka 分布式 & 集群相关的内容，如远程 Actor、远程路由、事件监听、集群搭建、集群分片、持久化等，相信大家通过书中的示例，可以很快搭建自己的分布式应用。

第 9 章主要介绍了如何使用 Akka 构建 HTTP 和 WebSocket 服务，并完整实现了一个图文聊天系统。

第 10 章主要介绍了一款“全家桶式”的微服务框架 Lagom，它是基于 Akka 和 Play 构建的，提供了一整套微服务解决方案。虽然在 Java 界已经有了 Spring Boot 或 Spring Cloud 这类“既叫好又叫座”的微服务框架，但是大家不妨也看看 Lagom，或许能带给你新的思路。

## 勘误与反馈

由于水平有限，写作时间仓促，书中难免会出现一些错误或不准确的地方，还请大家多多批评指正。假如你在阅读本书的过程中，有任何更好的意见或建议，希望能通过微信

仅供非商业用途或交流学习使用

duyunfeiRoom、邮箱 angel6380@126.com 与我联系。非常期待得到大家的反馈，多谢！

## 致谢

本书得以顺利出版，首先要感谢机械工业出版社的杨福川老师，正是因为他的信任，才让我能有机会完成本书。由于今年实在太过忙碌，导致中途屡次拖稿，但是每次都能得到福川老师的理解与鼓励，这使我备受感动！同时也感谢我的编辑小艺和李雷鸣，他们在审稿期间给了我很多好的建议，很佩服你们的“火眼金睛”！

另外，也感谢为本书写推荐序的多位技术专家们。平日里，我会经常阅读他们的文章、博客和公众号，也和他们进行过多次交流和讨论，总是受益匪浅，能邀请到他们，是我的荣幸！

写书的过程辛苦且漫长，很感谢家人、朋友的支持与陪伴，你们是我最大的动力！

杜云飞

2018 年 1 月于上海



仅供非商业用途或交流学习使用

# Contents 目　录


前言

**第 1 章　认识 Akka** ····· 1

1.1　Akka 简介 ····· 1

1.1.1　技术背景 ····· 1

1.1.2　Akka 是什么 ····· 3

1.2　Akka 应用场景 ····· 4

1.3　Akka 架构体系 ····· 4

1.3.1　Actor 模型 ····· 4

1.3.2　体系结构 ····· 4

1.4　本章小结 ····· 7

**第 2 章　走进 Actor** ····· 8

2.1　Actor 组件 ····· 8

2.1.1　Akka 中的 Actor ····· 8

2.1.2　ActorSystem 与监管 ····· 9

2.1.3　生命周期监控 ····· 9

2.1.4　引用与路径 ····· 10

2.2　Akka 环境搭建 ····· 10

2.2.1　环境准备 ····· 11

2.2.2　使用 Lightbend Activator 平台 ····· 12

2.3　创建一个 Actor ····· 14

2.3.1　定义 Actor ····· 14

2.3.2　创建 Actor 实例 ····· 14

2.3.3　工厂模式 ---Props/Creator ····· 15

2.4　发送 – 接收消息 ····· 16

2.4.1　tell 方法 ····· 16

2.4.2　ask 方法 ····· 17

2.4.3　消息转发 ····· 18

2.5　查找一个 Actor ····· 18

2.6　消息不可变 ····· 21

2.7　Actor 行为切换 ····· 21

2.8　Actor 生命周期 ····· 25

2.9　停掉一个 Actor ····· 26

2.10　监督与容错处理 ····· 29

2.11　Circuit Breaker（熔断） ····· 33

2.12　配置相关 ····· 35

2.13　本章小结 ····· 36

**第 3 章　线程调度** ····· 38

3.1　什么是 Dispatcher？ ····· 38

3.2　Executor 选择 ····· 39

3.3　配置 Dispatcher ····· 39

3.4　使用 Dispatcher ····· 41




3.5 PinnedDispatcher ····· 41
3.6 本章小结 ····· 42

**第 4 章 邮箱** ····· 43

4.1 消息处理顺序 ····· 43
4.2 默认邮箱配置 ····· 44
4.3 内置邮箱 ····· 45
4.3.1 内置邮箱说明 ····· 45
4.3.2 自定义优先级 ····· 45
4.3.3 控制指令优先 ····· 48
4.4 Actor 使用邮箱的多种方式 ····· 49
4.4.1 配置 Actor 邮箱 ····· 49
4.4.2 配置 dispatcher 邮箱 ····· 49
4.4.3 实现 RequiresMessageQueue 接口 ····· 50
4.5 自定义邮箱类型 ····· 50
4.6 本章小结 ····· 52

**第 5 章 消息路由** ····· 53

5.1 Router 和 Routee ····· 53
5.2 路由器及路由逻辑 ····· 53
5.3 路由 Actor ····· 56
5.3.1 Pool 方式 ····· 56
5.3.2 Group 方式 ····· 58
5.4 常见路由类型 ····· 59
5.4.1 广播 -Broadcast ····· 60
5.4.2 最快响应 -ScatterGatherFirst Completed ····· 61
5.4.3 随机 – 最快响应 -TailChopping ····· 62
5.4.4 创建可修改容量的池 ····· 62
5.5 特殊消息处理 ····· 63
5.5.1 Broadcast 消息 ····· 64
5.5.2 PoisonPill 消息 ····· 64
5.5.3 其他管理类消息 ····· 64
5.6 本章小结 ····· 65

**第 6 章 实用工具** ····· 67

6.1 定时调度——Scheduler ····· 67
6.2 处理并发结果——Future ····· 68
6.2.1 Future 常规用法 ····· 68
6.2.2 函数式 Future ····· 70
6.3 事件总线 ····· 72
6.3.1 实现事件总线 ····· 72
6.3.2 事件流处理 ····· 74
6.4 日志处理 ····· 75
6.4.1 基础配置 ····· 76
6.4.2 使用 slf4j ····· 76
6.5 Akka 扩展 ····· 78
6.5.1 TypedActor 初探 ····· 78
6.5.2 自定义扩展 ····· 81
6.5.3 集成 Spring ····· 83
6.6 Akka I/O ····· 87
6.6.1 TCP 服务 ····· 87
6.6.2 UDP 服务 ····· 90
6.7 Akka Streams ····· 91
6.7.1 Streams 组件 ····· 91
6.7.2 组合 Source、Sink ····· 94
6.7.3 案例：日志处理 ····· 95
6.7.4 错误处理 ····· 99
6.7.5 关联 Actor ····· 101
6.8 本章小结 ····· 102

**第 7 章 远程** ····· 103

7.1 远程介绍 ····· 103

7.1.1 Java RPC ····· 103
7.1.2 Akka 远程 ····· 106
7.2 创建远程 ActorSystem ····· 107
7.3 Actor 远程访问 ····· 108
7.4 创建远程 Actor ····· 109
7.5 远程路由器 ····· 110
7.6 远程事件 ····· 113
7.7 序列化 ····· 114
7.7.1 Protobuf 框架 ····· 114
7.7.2 序列化 API ····· 115
7.8 本章小结 ····· 116

**第 8 章 集群** ····· 117

8.1 Akka 集群概述 ····· 117
8.1.1 种子节点 ····· 118
8.1.2 领导节点 ····· 118
8.1.3 节点状态及生命周期 ····· 118
8.1.4 故障检测 ····· 119
8.2 创建 Akka 集群 ····· 119
8.2.1 代码及集群配置 ····· 119
8.2.2 启动集群 ····· 122
8.2.3 加入集群 ····· 123
8.2.4 akka-cluster 集群工具 ····· 123
8.3 集群示例——实现文章单词统计 ··· 125
8.3.1 示例分析 ····· 125
8.3.2 代码实现 ····· 127
8.4 使用路由 ····· 133
8.4.1 Group 路由 ····· 133
8.4.2 Pool 路由 ····· 133
8.5 集群指标 ····· 136
8.5.1 指标收集 ····· 136
8.5.2 基于指标的负载均衡 ····· 138
8.6 集群单例 ····· 139
8.7 集群客户端 ····· 140
8.7.1 集群客户端概述 ····· 140
8.7.2 使用 ClusterClient ····· 141
8.7.3 事件监听 ····· 142
8.8 集群分片 ····· 143
8.8.1 分片概念 ····· 143
8.8.2 持久化 Actor ····· 144
8.8.3 分片示例 ····· 148
8.9 本章小结 ····· 155

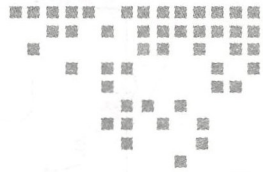
**第 9 章 HTTP 服务** ····· 156

9.1 HTTP 协议 ····· 156
9.1.1 header 信息 ····· 157
9.1.2 状态码 ····· 159
9.2 Akka HTTP ····· 159
9.2.1 配置依赖 ····· 159
9.2.2 HTTP 服务端 ····· 160
9.2.3 请求和响应 ····· 160
9.2.4 HTTP 客户端 ····· 163
9.2.5 使用 Routing DSL ····· 163
9.2.6 常用 Directive ····· 165
9.3 Akka WebSocket ····· 166
9.3.1 WebSocket 协议 ····· 167
9.3.2 Akka WebSocket ····· 167
9.3.3 WebSocket 应用之聊天室 ····· 169
9.4 本章小结 ····· 176

**第 10 章 微服务全家桶之 Lagom** ····· 177

10.1 Lagom 简介 ····· 177
10.2 Lagom 初探 ····· 178
10.3 搭建 Lagom 服务 ····· 181

10.3.1 项目结构……181
10.3.2 编写 Lagom 服务……182
10.4 持久化实体……186
10.4.1 持久化简介……186
10.4.2 PersistentEntity API……187
10.4.3 持久化示例……187
10.4.4 MySQL 支持……191
10.5 Lagom 部署（SandBox）……192
10.5.1 安装 ConductR SandBox……192
10.5.2 部署 Maven 项目……195
10.6 本章小结……199



第 1 章 Chapter 1

# 认识 Akka

打开本书第一章，你或许已经迫不及待地想体验一下“指尖”上的 Akka，撸起袖子手起刀落地写下第一个例子：Hello World。别急，在选择（或学习）一种技术前，理解 why 比 how 可能更加重要，而了解基本的背景非常有助于我们理解 why，即：我们为什么选择并相信它？

本章主要介绍 Akka 框架的技术背景，应用场景以及架构体系！

## 1.1 Akka 简介

当今社会，一方面，人们越来越享受互联网带来的种种便利，同时也对互联网产品有了更高的要求，比如更快的响应速度和更稳定的服务；另一方面，互联网产品在不断发展的过程中也面临着非常多的技术挑战，比如服务化、分布式、并行计算等，那么，Akka 在其中的哪些领域可以一展身手呢？

### 1.1.1 技术背景

在产品的早期阶段，由于用户量小，业务也比较单一，在开发时大多采用集中式的架构。在该架构下，工程师可以快速地进行产品迭代，而且由于此时的 Bug 潜伏期比较短，并且容易查找，整个项目的质量都处于可控状态。随着业务的发展，系统功能不断升级，产品逐渐变得复杂，新的开发需求也接踵而至，工程师除了要“如履薄冰”地添加新的功能，还要疲于应付系统潜在的 Bug（如图 1-1 所示），当这种情况持续一段时间后，系统会变得更加臃肿不堪且难以维护，工程师们也会在此时滋生更多抱怨，不利于团队稳定。

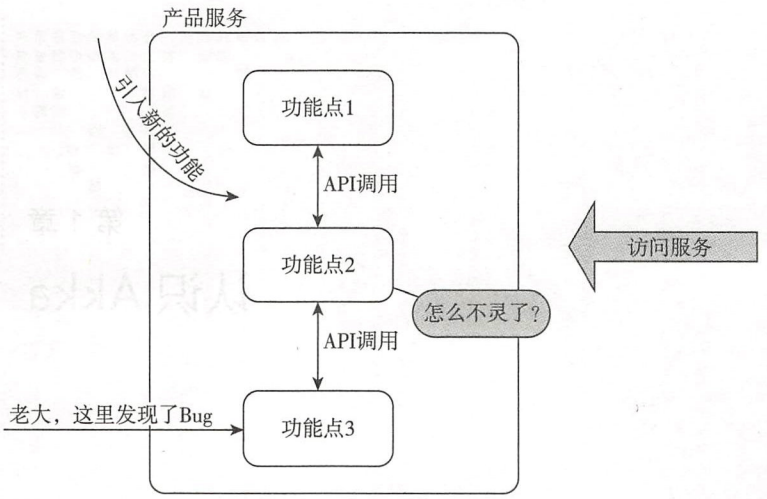



图 1-1　集中式架构的项目，新功能开发以及 Bug 维护让人头痛

在这个时候，架构师们会意识到，唯一能解决这个问题的良方是：拆。

一般情况下，我们会尝试把当前的系统拆成多个子系统并部署在不同服务器上，每个子系统尽量承担单一职责，这样既可以解决由于代码累积造成的维护困难，也有助于分散高并发流量带来的压力。

在拆的时候，我们也得知道，各个子系统之间往往存在某种关联性和依赖性，不太可能是孤立的，并且我们通常会把公共业务单独提取出来，希望它能作为独立的应用对外提供服务，即所谓的分布式架构，那么此时要解决的问题是：我们该怎样让分布式系统中各个子系统或子应用之间能够互相通信或调用。

拆分后的架构如图 1-2 所示。

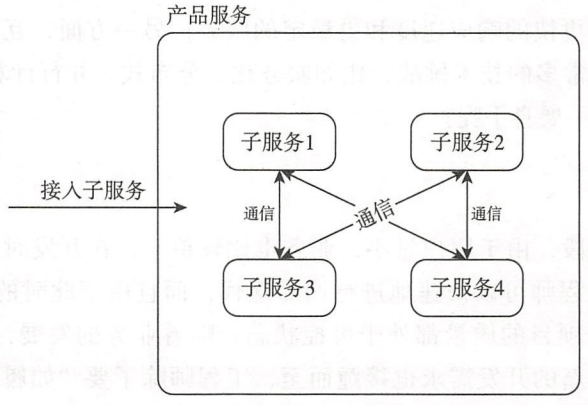



图 1-2　系统拆分后互相调用示意图

实际上，自己实现一个简易的远程通信（或调用）不是难事，比如基于 TCP Socket 或

HTTP 的 RPC 都可以实现，但是在实际项目中，事情远没有这么简单，比如异步处理、容错机制、扩展性、性能问题等都需要做周密的考虑，而 Akka 就天生拥有异步、分布式消息处理的能力，并且拥有强劲的容错机制和性能，可以帮助开发者解决这类问题。

在产品不断发展的过程中，另外一个不可忽视的挑战是大数据带来的存储和计算问题，存储暂且不表（Akka 不直接提供数据存储的方案，当然，在 Akka 的基础上也可以实现类似的分布式文件存储系统），我们来看看数据的计算问题。

在一个系统中，数据的来源非常广，比如用户交易数据、历史记录、系统日志等，当数据积累到一定程度后（变成 Big Data），我们就得考虑挖掘其价值了。

以电商为例，当用户打开浏览器输入域名并按下回车键的那一刻开始，系统就默默地在记录用户数据了，比如访问时间、终端设备类型、用户 IP 等；当用户看到某个商品并点击链接后，系统会收集并更新商品点击数（当然，也有可能是广告）；当用户满怀期望地加入购物车后，系统会把该商品记录到用户的“意向清单”中；当用户毫不犹豫地完成支付时，系统已经生成订单数据并考虑向你推荐其他匹配商品了（笑）。这是笔者虚拟出来的一个数据收集场景，在实际项目中，收集的数据维度只会比这更多，生成的数据量也会更大。很多大型的电商平台（比如亚马逊、淘宝等）都有自己的“个性化推荐系统”，该系统会根据大量的历史数据（比如浏览、购买、交易，评论等）预测（计算）出用户可能的喜好，以此做出合理的商品推荐，进而提升购买转化率。

面对大量的计算任务，系统怎样才能快速实时地得到想要的结果呢？很显然，依靠单核 CPU 的处理能力已不足以进行如此密集的计算（摩尔定律的失效），一般情况下，我们的解决方案是：把计算拆分成多个子任务实现并行（单机多核或分布式集群）执行。在 Java 中，我们可以通过 Fork/Join 等框架来实现单机的并行程序，但是假如想要在单机或分布式以相同的模式去处理计算，很明显这些框架是远远不够的，除此之外，我们还必须考虑多节点计算时的容错、分布式通信等问题。

由此可见，要从头实现这样一个解决方案并非易事，很多时候，我们希望选择一款设计精良，易于使用的框架来处理这些问题，而 Akka 正是为此而生！

### 1.1.2　Akka 是什么

Akka 是一款高性能、高容错性的分布式 & 并行应用框架，遵循 Apache 2 开源许可，底层通过 JVM 上另外一个流行的语言 Scala 实现，提供 Java&Scala API（本书采用 Java 进行代码演示）。它基于经典的 Actor 并发模型（即所有的消息都是基于 Actor 组件进行传递，后面我们会做详细介绍），拥有如下特点：

- **并行与并发**：提供对并行与并发的高度抽象。
- **异步非阻塞**：Akka-Actor 消息通信都是基于异步非阻塞。
- **高容错性**：为跨多 JVM 的分布式模型提供强劲的容错处理，号称永不宕机。
- **持久化**：Actor 携带的状态或消息可以被持久化，以便于在 JVM 崩溃后能恢复状态。

❑ **轻量级**：每个 Actor 大约只占 300bytes，即 1G 内存可容纳接近 300 万个 Actor。

基本上，Akka 从底层就解决了我们大多数分布式 & 并行程序常见的难题，让工程师更专注于业务实现，同时，它也保留了多个扩展接口及配置，便于满足个性化定制的需要！

## 1.2 Akka 应用场景

目前 Akka 已经在多家互联网 & 软件公司广泛使用，比如 eBay、Amazon、VMWare、PayPal、阿里、惠普、豌豆荚等，涉及行业包括游戏、金融投资、医疗保健、数据分析等。

使用场景包括：

❑ 服务后端：比如 rest web，websocket 服务，分布式消息处理等。

❑ 并发 & 并行：比如日志异步处理，密集数据计算等。

总之，对高并发和密集计算的系统，Akka 都是适用的！

## 1.3 Akka 架构体系

Akka 采用 Scala 开发，运行于 JVM 之上，提供了 Scala 和 Java 两种 API，目前属于 Lightbend 公司（原名 Typesafe）。它实现了经典的 Actor 模型，同时也提供了丰富的组件，比如邮箱（MailBox）、路由（Routing）、持久化（Persistence）、网络（包括远程、集群）等，在底层对分布式 & 并行模式进行了高度且统一的抽象，使工程师用很少的代码就可以实现一个完整的分布式应用。本节我将为大家介绍 Akka 的整个体系结构以及相关概念。

### 1.3.1 Actor 模型

Actor 模型最早在 1973 年由 Carl Hewitt 提出，它高度抽象了分布式并行程序的运行模式，从底层屏蔽了线程和锁机制的管理，为开发者提供了简单可依赖的开发方式。

Actor 模型认为，并行计算的最小单元就是一个 Actor 实例，而每个实例拥有自己的状态和行为，在一个大型系统中，可能存在成千上万个 Actor 实例，它们之间通过消息的方式进行通信，每个 Actor 都能发送消息给其他 Actor，也能从其他 Actor 接收消息。当我们在执行某个计算任务时，会给对应的 Actor 实例发送一个相关的消息，该 Actor 在接收消息后开始执行计算任务，由于整个消息通信的过程是异步的，所以不用等到 Actor 执行完整个过程就能执行下一步（发送消息后会马上返回），这种异步通信的方式大大提高了程序的响应性。

Actor 异步消息通信流程如图 1-3 所示。

### 1.3.2 体系结构

Actor 是 Akka 最核心的概念，也是最基本的执行单元，所以对 Actor 管理和监控的有

效性是极为重要的。在 Akka 中，每个 Actor 都有自己的监管对象，即该 Actor 的创建者，它们通常会负责子 Actor 的失败处理，另外，某些 Actor 也需要对生命周期进行监控（比如该 Actor 的终止），以便及时响应并做正确处理，这些监督和监控者本身也都是一个 Actor。在 Akka 中，整个 Actor 体系被抽象成一个 ActorSystem，它是一个层级的结构，拥有公共行为的配置和管理。

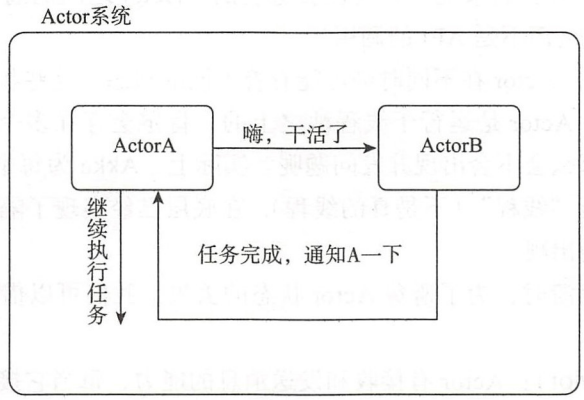



图 1-3　Actor 的异步消息通信流程

在 ActorSystem 基础上，Akka 也提供了一些配套的组件，比如持久化，HTTP 服务，网络服务等，它们都是构建高可用分布式应用不可或缺的部分，基本架构体系和周边产品如图 1-4 所示。

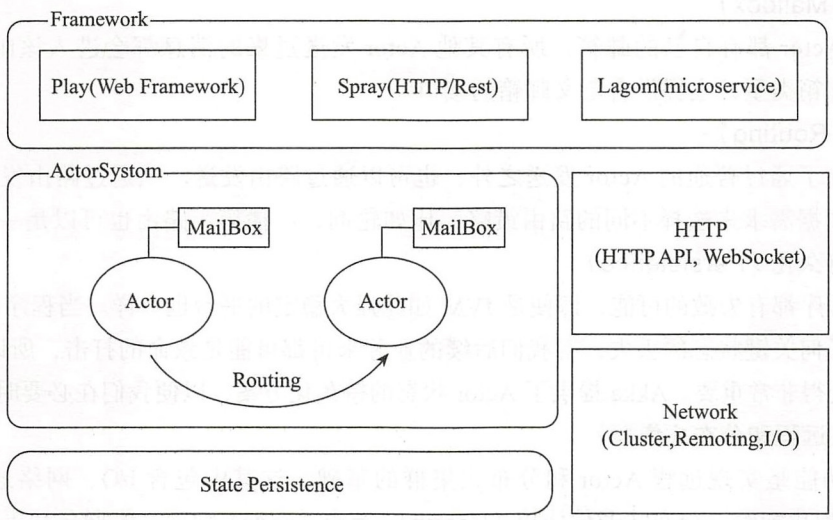



图 1-4　Akka 基本架构体系及周边产品

### Actor 组件

在 Akka 中，Actor 是一个高度抽象的对象引用，它包含以下几个要素：

- **引用（Actor Reference）**：Actor 的引用不同于普通对象的引用，也不能通过传统“new”的方式直接创建一个 Actor 对象。很多时候需要通过 actorOf 或者 actorSelection 等方式返回一个 ActorRef 对象，该对象有可能存在于本地，也可能存在于远程节点，对我们来说，它是位置透明的。Actor 之间的通信和执行任务都是通过消息驱动的（而不是 API 的调用）。
- **状态（State）**：Actor 在不同时刻可能有着不同的状态，这些状态用变量来表示。在底层实现上，Actor 是运行于线程池之上的，肯定会存在多个 Actor 共享同一个线程的情况，那么会不会出现并发问题呢？实际上，Akka 为每个 Actor 都抽象出一个轻量级的执行“线程”（不是真的线程），在底层已经实现了隔离，所以基本上不用担心该问题的出现。

另外，当 JVM 崩溃时，为了避免 Actor 状态的丢失，我们可以借助持久化方案来对状态进行持久化操作。

- **行为（Behavior）**：Actor 有接收和发送消息的能力，每当它接收一条消息后，就可以执行某个业务操作，同时也可以把消息转发到其他节点进行处理。
- **监管策略（Supervisor Strategy）**：Actor 系统是一个层级结构，当任务被某个 Actor 分摊到子 Actor 时，父 Actor 就拥有监管子 Actor 的义务。在监管时，我们需要根据不同的情况选择不同的处理方案（比如停止、重启、恢复或者失败上溯）和策略（比如 1 vs 1、1 vs *N* 策略）。

### 邮箱（Mailbox）

每个 Actor 都有自己的邮箱，所有其他 Actor 发送过来的消息都会进入该邮箱。Akka 自带多种邮箱类型，也提供自定义邮箱的接口。

### 路由（Routing）

消息除了通过普通的 Actor 发送之外，也可以通过路由发送。当通过路由发送消息时，我们可以根据需求来选择不同的路由策略，比如轮询、广播等。路由也可以是一个 Actor。

### 状态持久化（Persistence）

任何程序都有失败的可能，即便是 JVM 如此强大稳定的平台也一样。当程序出错、JVM 崩溃时，任何关键状态的丢失，对我们后续的业务来讲都可能是致命的打击，所以状态数据的持久化变得非常重要。Akka 提供了 Actor 状态的持久化方案，以便我们在必要时恢复数据。

### 网络（远程和分布式集群）

网络功能是实现远程 Actor 和分布式集群的基础，这其中包含 I/O、网络通信（TCP/UDP）、序列化配置、分布式通信协议（Gossip）、节点（node）管理、集群分片等内容。

### HTTP 模块

Akka 提供了简单易用的 HTTP 模块，支持完整的 HTTP 服务端与客户端开发，可以帮

助我们快速构建性能极强的 Rest Web 服务。

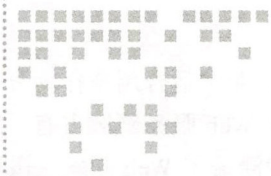
**相关开源项目**

Akka 具有高性能、可扩展、设计友好等诸多优点，非常适合用来作为分布式应用的基础框架，而且由于对 HTTP 有非常好的支持，也让它在 Web 服务领域占有一席之地。目前业界已经有多个基于 Akka 实现的开源项目，项目类型涵盖了 Web 开发、微服务、分布式文件或计算服务等。下面是 Akka 中两个具有代表性的开源项目：

- **Play 框架：**一款大名鼎鼎的 Web 开发框架。它不同于其他 Servlet 系的框架，比如 Struts 或者 SpringMVC，底层基于 Akka 构建，天生拥有异步的特点，具有极佳的性能。它默认提供 RESTful 风格的 API，同时也对 WebSocket 有不错的支持。
- **Lagom 框架：**在目前 IT 界，最火爆的概念要属“微服务”了，微服务的理念是，把业务功能拆成小的、独立的单元，它们之间能够互相通信而且支持水平扩展。Lagom 就是这样一款微服务框架，它基于异步的消息驱动，对分布式集群、持久化（如 JPA、NoSQL）都有良好的支持。同时，它也拥有完整的集成开发环境，非常便于在线部署和管理。

## 1.4　本章小结

随着互联网的高速发展，开发分布式 & 并行应用将不再是某些巨头公司的“专利”，而对大部分研发团队来讲，从零开发自己的分布式架构又会面临诸多挑战，此时选择 Akka 作为分布式并行服务的基础框架将是极好的！ Akka 底层采用 Scala 语言实现（JVM 上另外一款明星语言），它基于 Actor 模型，在底层帮助开发者屏蔽了异步、消息通信、容错处理、网络服务、分布式集群等实现细节。在 Akka 基础上，也诞生了 Play、Lagom 等应用框架，让开发者更容易打造自己的高可用分布式系统。在下一章，我们将正式走进 Akka-Actor 的世界，去深入了解 Actor 开发的所有细节，Let’s go!



Chapter 2 第 2 章

# 走进 Actor

开发一个功能完备的并行 & 分布式程序并不容易，我们需要考虑其中可能出现的任何问题。对于有经验的工程师来说，抽象出一个简单的并行模型也不算太复杂，我们使用 JDK 内置的并发包就可以轻松实现，但是这种做法大多在单机中运行良好，在分布式环境中可能并不适用，比如单机上的任务大多靠直接调用 API 来实现，而分布式环境下需要通过网络通信（通知）实现；单机程序中要得到一个返回结果，直接通过返回值得到，而分布式环境下需要通过消息回传来实现，诸如此类，在单机上的某些概念并不能无缝的推及到分布式环境中。

而 Akka 通过 Actor 模型很好地解决了这一问题，它认为，每个执行单元都应该是一个 Actor，他们都有着自己的使命，并且分布在网络中的任何一个节点，在执行某个任务时，我们无须关心每个 Actor 的具体位置，无论在单机还是分布式环境下，它们都有着统一的操作方式。

## 2.1 Actor 组件

### 2.1.1 Akka 中的 Actor

在 Akka 中，消息在不同 Actor 之间进行传递和处理，以此来驱动任务的执行，这和普通 OOP 的调用方式有明显的区别，在执行某个功能时，不会直接通过对象调用某个业务方法来实现。Actor 提供良好易用的“问答”式 API 来进行消息的通信，并且理论上不限制任何类型的消息，它们可以是一个字符串，也可以是一个对象，但是为了保证消息的状态不受意外修改，最好使用不变类型的对象。

除此之外，Actor 也拥有线程安全和轻量级的特点。

- **线程安全**：Actor 运行于线程池之上，但是几乎不用考虑它的安全性问题，因为单个 Actor 总是线程安全的，并且本身在处理接收到的消息时是串行的。
- **轻量级**：在一个大型的应用中，可能同时运行着成千上万个 Actor，这样会不会耗费太多资源？这点我们大可放心，在 Akka 中，每个 Actor 只占用 300 字节左右，即使单机内存不够用了，我们也可以很方便地切换成分布式模式。

## 2.1.2　ActorSystem 与监管

在一个应用中，所有的 Actor 共同构成了 Actor 系统，即 ActorSystem。它是一个层级结构，该结构已经限定了我们对某个 Actor 该施行怎样的管理策略，一般来讲，我们所用的 Actor 都可能有一个显式的父级 Actor（除了顶级 Actor 外），它会处理所有自己子孙失败的情况，比较常见的场景是：当子 Actor 在处理消息时出现了异常情况，父 Actor 可以通过匹配预先设定的某个动作来处理子 Actor，处理方式有：恢复子级、重启子级、停止子级、扩大化失败，这就是所谓的“父监督”模式。

那么父子关系是怎样定的呢？很简单，Actor 被谁创建，谁就是父级。实际上，在 ActorSystem 的创建过程中，会默认启动三个比较顶级的 Actor，而我们应用程序里面创建的 Actor 只是其中的一个分支而已，如图 2-1 所示。

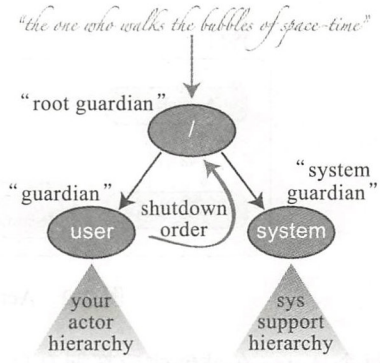



图 2-1　ActorSystem 层级和监管结构（摘自官网）

从图 2-1 中我们看到，Root guardian 是整个 ActorSystem 的根，它下面有两个分支，左边的 /user 分支是我们最常见到的，所有通过 ActorSystem.actorOf() 方法创建的 Actor 都属于该分支下，这是我们能手动创建的最高级别 Actor，其他通过 ActorContext.actorOf() 方法创建的 Actor 都是其子级（ActorContext 即 Actor 的上下文对象，后面会多次使用到），而右边的 /system 分支下的 Actor 都是系统层面创建的，主要与系统整体行为有关，在开发阶段并不需要对其过多关注。

在使用 ActorSystem 时，我们也需要知道，它是一个非常重量级的对象，一般来讲，每个应用程序只需要一个 ActorSystem 对象。

## 2.1.3　生命周期监控

每个 Actor 都会经历“生老病死”的阶段，在任何时候，我们都需要谨慎的对待它每个阶段可能出现的状况或者需要执行的任务。Actor 提供了生命周期管理的 API，比如 preStart、preRestart、postRestart、postStop 等，可以让我们知道，当前这个 Actor 到底处于什么状况，是启动？重启？还是已经停止了。

当一个 Actor 停止时，有时候我们希望释放掉不需要的资源，但有时候，我们得为此做出一些更复杂的或者更全局的响应，此时引入一个新的 Actor 作为“死亡监控者（Death Watch）”是有必要的，当某个 Actor 被停止后，监控者（其实是另外一个 Actor）可以收到一个 Terminated 消息，通过匹配该消息，监控者来决定到底该怎样做善后工作，这就是 DeathWatch 的做法。和上面讲的父监督比较类似，DeathWath 其实也描述了两个 Actor 之间的关系，但是这种关系可以通过 unwatch 的方式解除，而父子关系从出生那一刻就决定了，嗯，这真的非常拟人化！

### 2.1.4 引用与路径

Actor 不同于一般的 Java 对象，它既可能存在于本地，也可能存在于远程，但是对使用者来讲并没有明显的区分，仅仅只需要操作它的引用对象就行了，在 Akka 中，使用 ActorRef 来表示一个 Actor 的引用。ActorRef 对原生的 Actor 实例做了良好的封装（也可以理解为一种代理），外界不能随意修改其内部状态，并且只能通过统一的方式去操作一个 Actor。

ActorRef 和 Actor 实例的关系如图 2-2 所示。

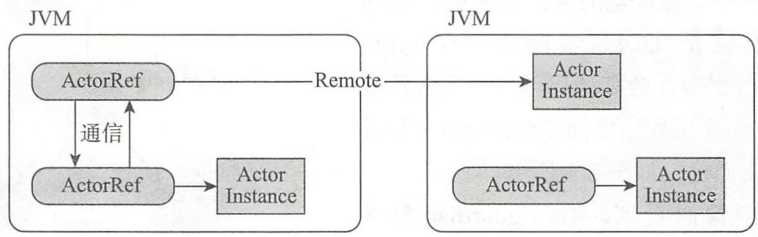



图 2-2 ActorRef 和 Actor 实例的引用关系

每个 Actor 在被创建后都拥有自己的路径，该路径遵循 ActorSystem 的层级结构，看起来是这个样子：

```
akka://mysys/user/parentActor/childActor
akka.tcp://mysys@127.0.0.1:2554/user/parentActor/childActor
```

其中 mysys 是 ActorSystem 的名字，/user 是用户创建 Actor 的默认根节点，parentActor 和 childActor 分别是父子 Actor，如果是远程 Actor，则需要给出 IP 和端口。通过路径，我们可以很方便地查找到一个 Actor。

## 2.2 Akka 环境搭建

通过 2.1 节的介绍，相信大家对 Akka-Actor 已经有了初步的认识，同时也对如何编写 Akka 应用产生了浓厚的兴趣。在本节中，笔者将介绍如何搭建 Akka 开发环境，以及怎样使用 Lightbend Activator 平台创建并演示示例程序，以便大家能顺利进入实战之旅！

## 2.2.1　环境准备

### JDK 选择

Akka 依赖 JVM 实现，所以我们必须首先安装 JDK。JDK 版本最好是在 1.6 以上，本书为了照顾到后面章节中所讲的新特性，所以采用 1.8 的版本。

### 开发工具

在开发工具方面，大家可以选择任何主流的 IDE，比如 Eclipse、IntelliJ IDEA 等。本书采用的开发工具是 Eclipse（JavaEE 版）。

### Maven 安装

Akka 支持通过 Maven、Gradle、SBT 来构建程序，本书采用 Maven 作为项目构建工具。下面简单介绍一下 Maven 的安装过程。

1）下载安装包：首先去官网 http://maven.apache.org/ 下载 Maven 压缩包，然后解压缩到一个非中文目录下，如笔者的解压目录为：C:\dvptool\apache-maven-3.0.4

2）配置 home 变量：新建环境变量 M2_HOME，并设置值为 C:\dvptool\apache-maven-3.0.4，如图 2-3 所示。

3）配置 path 变量：打开 path 变量，附加 ;%M2_HOME%\bin；如图 2-4 所示。




图 2-3　配置 Maven 的 home 变量

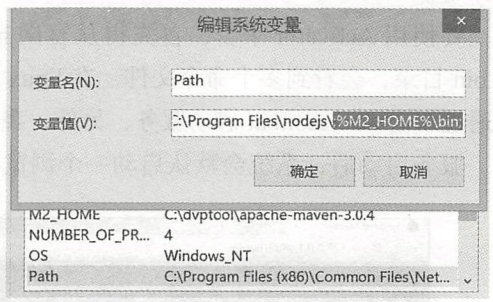



图 2-4　配置 path 变量

4）测试安装：进入命令行输入 mvn -v，会出现版本标识，表示安装成功，如图 2-5 所示。

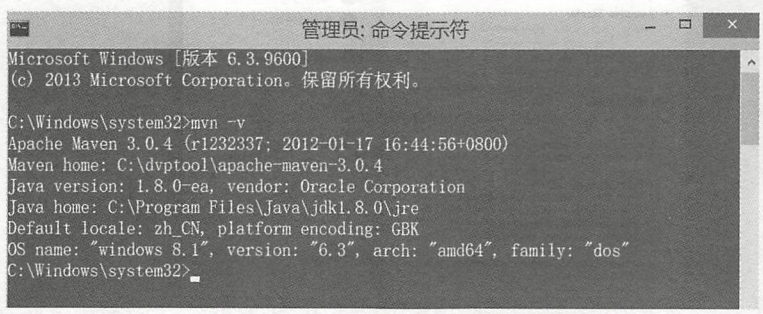



图 2-5　输入 mvn -v 得到版本信息

至此，maven 已安装成功。

**新建项目**

通过 Eclipse 新建一个 Maven Project，并且在 pom 中加入如下依赖：

```
<dependency>
<groupId>com.typesafe.akka</groupId>
<artifactId>akka-actor_2.11</artifactId>
<version>2.4.7</version>
</dependency>
```

Akka 把所有功能划分成了不同模块，这里我们仅仅配置了 Akka 核心模块：akka-actor，通过它已经可以实现基本的功能了。除了核心模块之外，它还包含了 akka-persistence、akka-remote、akka-cluster 等模块，分别用于实现持久化、远程、集群等功能。

更多模块的使用将在后续章节为大家做详细的介绍。

## 2.2.2 使用 Lightbend Activator 平台

Lightbend Activator 是 Lightbend 公司维护的一款应用开发平台，它提供了非常易用的 Web 操作界面，让程序员能快速进行项目构建，它内置支持的项目类型有 Scala、Akka、Play 等。

要使用 Activator 平台，首先得从官网 http://www.lightbend.com/ 下载安装包，解压后进入 bin 目录，会看到多个命令文件，在 linux 上可以通过 activator 启动服务，在 windows 上可以通过双击 activator.bat 启动服务，另外，我们也可以通过命令 activator ui 来启动服务。

服务启动后，系统会默认启动一个浏览器进程，并显示管理界面，如图 2-6 所示。

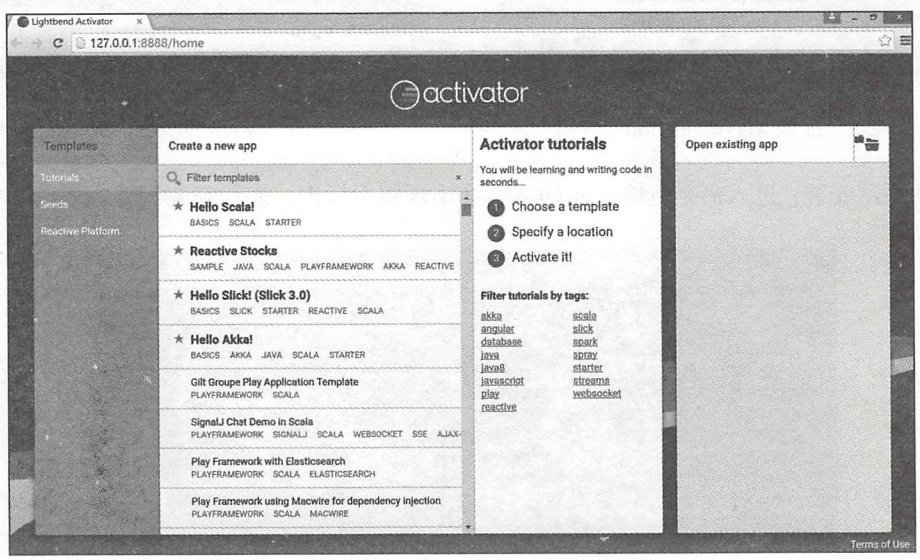



图 2-6 Lightbend Activator 平台管理界面

该平台已经内置了多个示例项目，点击其中任意一个，就可以看到项目描述和创建按钮（Create App）。

这里我们找到 Akka 的入门示例程序“Hello Akka”，点击 Create App，此时会在本地生成一个完整的 Akka 项目。创建完毕之后进入该项目，可以看到左边有 Build、Code、Run、Test 等目录，它们分别提供编译、源码浏览、运行、测试等功能，如图 2-7 所示。

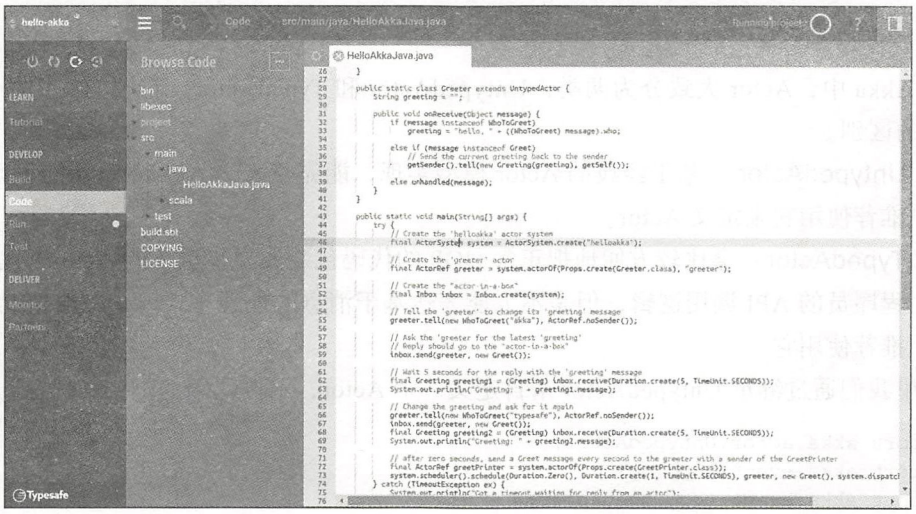



图 2-7　Lightbend Activator 示例程序操作界面

在 Code 里面，我们不仅可以查看源码，还可以对其进行修改。假如要执行程序，可以进入 Run 菜单，并点击上面的 Run/Stop 按钮，如图 2-8 所示。

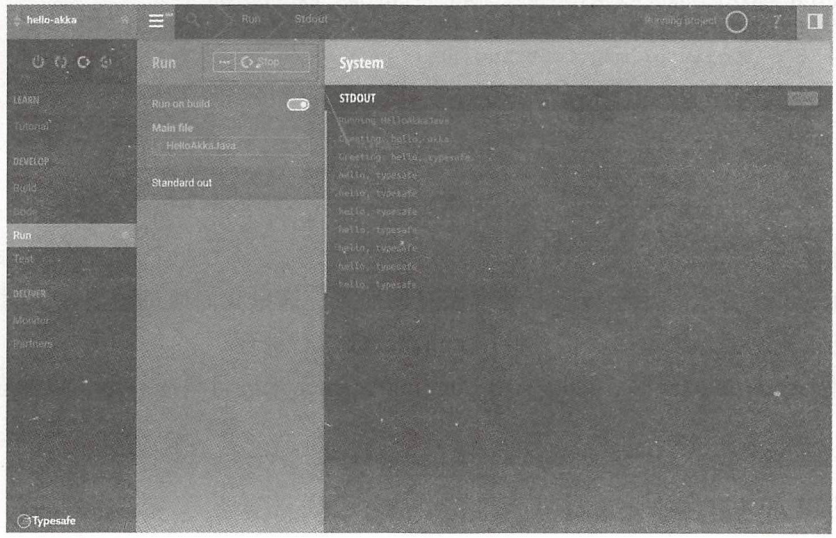



图 2-8　运行程序，得到输出结果

通过 Lightbend Activator 平台，我们可以很快速地搭建一个 Akka 项目，但是这些项目毕竟只是模板性质的，所以仅仅比较适合用来当作学习素材。

## 2.3 创建一个 Actor

### 2.3.1 定义 Actor

在 Akka 中，Actor 大致分为两类：UntypedActor 和 TypedActor，它们的使用场景有比较明显的区别。

- UntypedActor：基于经典的 Actor 模型实现，能完整表达 Akka-Actor 的设计思想，推荐使用它来定义 Actor。
- TypedActor：会比较方便地把正常 OOP 的代码包装成异步执行的 Actor，比较符合程序员的 API 调用逻辑，但是不太能表达基于消息的处理方式，在一般情况下，不推荐使用它。

这里我们通过继承 UntypedActor 来自定义一个 Actor，示例代码如下：

```
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

public class ActorDemo extends UntypedActor {

private LoggingAdapter log =Logging.getLogger( this .getContext().system(), this);

@Override
    public void onReceive(Object msg ) throws Exception {
        if (msg instanceof String) {
            log.info( msg.toString());
        } else {
            unhandled( msg);
        }
    }
}
```

代码解释：

1）onReceive 方法是用于接收并处理消息的地方，这里我们通过类型做了简单的匹配，当匹配不到相应的消息类型时，推荐使用 unhandled 进行处理；

2）在做 Actor 的调试时，可以采用 LoggingAdapter 来进行日志打印，以便获得更详细的信息。

### 2.3.2 创建 Actor 实例

我们看到，自定义一个 Actor 如此的简单，那么怎样使用它呢？正常情况下，我们通过

new 的方式就可以得到一个对象的引用，然后通过该引用来调用相应的 API，如下所示：

```
ActorDemo ad =new ActorDemo();
```

当我们执行这行代码时，系统会抛出如下异常：

```
Exception in thread "main" akka.actor.ActorInitializationException : You cannot
    create an instance of [com.javaakka.utpactor.ActorDemo] explicitly using the
    constructor (new). You have to use one of the 'actorOf' factory methods to
    create a new actor
```

很明显，这里不能通过 new 的方式得到一个 Actor 的引用。这正是 Akka 的有意为之，笔者之前介绍过，Actor 作为核心执行单元，天生拥有并行运行和分布式的特点，为了屏蔽实现细节（包括对内部状态的封装），简化调用方式（特别是位置透明性），统一资源调配和层级管理，我们只能通过 Akka 提供的某些 API 才能创建或查找 Actor。

一般来说，我们会通过 ActorSystem 或 ActorContext 来创建 Actor：

```
//参数为ActorSystem的名字，可以不传
ActorSystem system =ActorSystem. create( "sys" );
//参数分别是构造器和Actor的名字，名字可以不传
ActorRef actorRef =system.actorOf(Props.create(ActorDemo.class), "actorDemo" );
```

代码解释：

1）ActorSystem 是一个比较重量级的对象，一般来说每个应用程序只需要创建一个该对象；在创建 ActorSystem 和 Actor 时，最好都给出名字，在该示例中，它们的名字分别是 sys 和 actorDemo，同时要注意，在同一个 ActorSystem 中 Actor 不能重名；

2）actorOf 返回的不是 Actor 本身，而是 ActorRef，即 Actor 的引用，我们就是通过该引用来进行消息通信的；

3）假如 Actor 的构造函数有参数，可以通过 create 方法传入。

通过 ActorSystem 创建的是一个顶级的 Actor（即 /user 路径下），但是在实际项目中，我们可能希望创建具有层级关系的 Actor 网络，然后把任务交给子 Actor 去处理，并且让父级去监督子级。

此时我们可以使用 Actor 中的 getContext 来创建子 Actor：

```
ActorRef childActor=getContext().actorOf(Props.
create(ChildActor.class),"childActor" );
```

### 2.3.3　工厂模式 ---Props/Creator

ActorSystem 和 ActorContext 通过接收一个 Props 实例来创建 Actor，而 Props 实例本身有两种方式可以创建：

- 指定 Actor 的 Class，比如：Props.create（ActorDemo.class);
- 指定一个 Actor 工厂，实现 akka.japi.Creator 接口，重写其 create 方法。

对于比较简单的场景，直接使用第一种即可，而对于需要统一配置或创建某个 Actor 的

场景，则需要自己实现 Creator 工厂接口，一般来讲，我们会把该工厂直接定义在 Actor 内部，作为静态 API 存在，下面是示例代码：

```
class PropsDemoActor extends UntypedActor{
@Override
public void onReceive(Object msg) throws Exception {
}
public static Props createProps() {
    //实现Creator接口并传入Props.create方法
    return Props.create(new Creator<PropsDemoActor>(){

    @Override
    public PropsDemoActor create() throws Exception {
    //创建Actor
        return new PropsDemoActor();
    }
});
}}
```

此时创建该 Actor 的方式如下：

```
ActorRef ref = system.actorOf(PropsDemoActor.createProps(),"propsActor");
```

有些时候，我们可能会把生成的 Props 实例当作消息传给其他 Actor，让对方“帮”我们创建符合要求的 Actor 实例，真正做到“一次模板，到处创建”！

## 2.4 发送 – 接收消息

创建完 Actor 后，我们就可以通过 ActorRef 对象和 Actor 进行消息通信了。在 Akka 中，可以通过两种方式给一个 Actor 发送消息，tell 和 ask。它们都以异步的方式发送消息，不同的是，前者发完后立即返回，而后者期待得到一个返回结果，假如在设置的时间 (Timeout) 内没有得到返回结果，发送方会收到一个超时异常。另外，任何给 Actor 发送的消息都会通过 receive 方法收到。

### 2.4.1 tell 方法

假如仅仅需要异步的给某个 actor 发送消息，可以调用 tell 方法，调用方式如下：

```
actorRef.tell( "Hello Akka",ActorRef.noSender());
```

第一个参数即“消息”，它可以是任何可序列化的数据或对象，第二个参数表示发送者，通常来讲是另外一个 Actor 的引用，noSender() 表示无发送者（实际上是一个叫做 deadLetters 的 Actor，后面 6.3.2 节会再提到）。假如想在 Actor 内部得到发送者，可以调用 getSender() 方法。

在调用 tell 方法后，Actor 就会异步的去处理该消息，并不会阻塞后续代码的运行。

### 2.4.2　ask 方法

当我们需要从 Actor 里面得到一个返回结果时，使用 ask 是比较方便的，这是一种典型的“请求 - 响应”模式，比较符合我们常见的调用思维。ask 方法会将返回结果包装在 scala.concurrent.Future 中，此时你可以“就地”（阻塞）获取这个值，也可以通过异步回调函数获取这个值，考虑到性能问题，我们往往会采用后者。

为了能将数据返回给发送方，我们需要先引用到 Sender，如下面代码所示：

```
public class AskActorDemo extends UntypedActor {
    @Override
    public void onReceive(Object msg) throws Exception {
        System.out.println( "发送者是: " +getSender());
        getSender().tell( "hello " +msg , getSelf());
    }
}
```

在 onReceive 方法中，我们通过 getSender() 引用到发送方 Actor，然后将消息发送过去，并把当前 Actor 设置为 Sender（注意：这里不是 this）。Ask 方法由 akka.pattern.Patterns 提供，它需要传入一个超时时间，以便进行返回超时的处理，下面是示例代码：

```
ActorRef ask_ar =system .actorOf(Props.create(AskActorDemo. class),
"askActorDemo");
    Timeout timeout= new Timeout(Duration.create(2, "seconds"));
    Future<Object> f =Patterns.ask( ask_ar, "Akka Ask" ,timeout );
    System.out.println("ask..." );
    f .onSuccess( new OnSuccess<Object>() {
    @Override
    public void onSuccess(Object result) throws Throwable {
        System. out .println("收到消息: " +result );
    }
}, system.dispatcher());
System. out.println( "continue..." );
```

执行该程序，可以得到如下结果：

```
ask...
continue...
发送者是: Actor[akka://sys/temp/$a]
收到消息: hello Akka Ask
```

**代码解释：**

Patterns.ask 方法会异步执行，它的参数分别是：Actor 引用，发送的消息，超时时间。假如 actor 返回消息超时了，会产生一个 akka.pattern.AskTimeoutException；Patterns.ask 方法返回一个 Future 对象，它提供了 onSuccess 和 onFailure 两个回调函数，以便我们处理返回数据和异常信息。除此之外，Future 还拥有更多处理结果的方式，比如合并、组合等，笔者将在 6.2 节对它做更详细的介绍。

### 2.4.3 消息转发

除了直接给某个 Actor 发送消息外，我们还可以通过 forward 方式对消息进行转发，比较典型的使用场景是实现消息路由及负载等功能，下面我们来看看，怎样实现一次消息转发。

首先定义一个 Actor，作为目标接收者：

```
class TargetActor extends UntypedActor {

    @Override
    public void onReceive(Object msg) throws Exception {
        System.out.println("TargetActor receive: " + msg + ", sender=" + getSender());
    }
}
```

然后定义转发者：

```
class ForwardActor extends UntypedActor {
    private ActorRef target =
    getContext().actorOf(Props.create(TargetActor.class),
    "targetActor");

    @Override
public void onReceive(Object msg) throws Exception {
        target.forward(msg, getContext());
    }
}
```

此时我们给 ForwardActor 发送的消息（代码略），会通过 target.forward(msg, getContext()) 转发给 TargetActor，并且 sender 仍然是 ForwardActor 的发送者（而不是 ForwardActor 本身），实际上，forward 方法和 target.tell( msg,getSelf()) 实现的功能类似。

## 2.5 查找一个 Actor

对于已存在的 Actor，我们可以根据路径来进行查找，API 如下：

```
ActorSelection=[ActorSystem/ActorContext ].
actorSelection([path]);
```

在调用它时，必须指定一个绝对或者相对路径（包括远程路径），比如：

```
/user/parentActor/childActor
../parentActor/childActor
akka.tcp://mysys@127.0.0.1:2554/user/ parentActor /childActor
```

actorSelection 方法会返回一个 ActorSelection 对象，它并不等同于 ActorRef，但是你几乎可以使用同样的方式（tell 或 ask）来与 Actor 进行通信，当所查路径不存在时，消息肯

定也就不会发送成功。假如想得到目标 ActorRef，需要向 ActorSelection 发送一个 Identify 消息，这个消息是内置的，所有 Actor 都能识别它，然后对方会自动返回一个 ActorIdentity 对象，该对象里面包含了 ActorRef 信息，这个过程如图 2-9 所示。

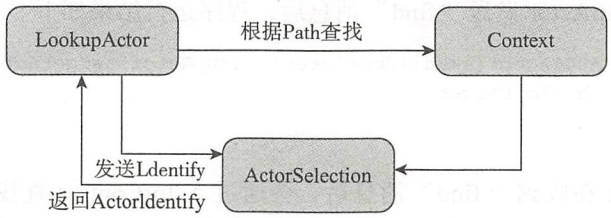



图 2-9　根据 path 查找 Actor 并得到 ActorIdentify（包含 ActorRef）

下面我们用代码来实现这一过程，首先新建目标 Actor，取名为 TargetActor，表示被查找的对象：

```
class TargetActor extends UntypedActor {
@Override
    public void onReceive(Object msg) throws Exception {
        System.out.println("target receive: "+msg);
    }
}
```

然后新建用于查找的 Actor，取名为 LookupActor，为了让程序更加精简，这里直接用 LookupActor 创建 TargetActor，具体代码如下：

```
class LookupActor extends UntypedActor {

    private ActorRef target = null;
{
        target = getContext().actorOf(Props.create(TargetActor.class), "targetActor");
}
@Override
public void onReceive(Object msg) throws Exception {
    if (msg instanceof String) {
        if ("find".equals(msg)) {
            ActorSelection as = getContext().actorSelection("targetActor");
            as.tell(new Identify("A001"), getSelf());
        }
    } else if (msg instanceof ActorIdentity) {
        ActorIdentity ai = (ActorIdentity) msg;
        if (ai.correlationId().equals("A001")) {
            ActorRef ref=ai.getRef();
if(ref!=null) {
            System.out.println("ActorIdentity: " + ai.correlationId() + ""+ ref);
            ref.tell("hello target",getSelf());
        }
}
```

```
        } else {
            unhandled(msg);
        }
    }}
```

当我们向 LookupActor 发送“find”消息后，程序运行结果如下：

```
ActorIdentity: A001Actor[akka://sys/user/lookupActor/targetActor#1316975245]
target receive: hello target
```

**代码解释：**

- LookupActor 在收到“find”消息后，会通过 ActorContext 查找出 ActorSelection 对象，由于该 TargetActor 对象是当前 Actor 的子级，所以可以直接指定“targetActor”而不用绝对路径，“targetActor”的绝对路径是：/user/lookupActor/targetActor；
- 给 ActorSelection 发送 Identify 时，需要指定一个 messageId，用来区分所查找的不同 Actor（可能会查找多个 Actor，后面需要通过 messageID 来判断）。消息发送后，当前 Actor 会自动收到一个 ActorIdentity，通过该对象的 getRef 方法得到 ActorRef，这里要注意的是，ActorSelection 并不能保证 Actor 一定存在，所以我们通常会对 ActorRef 做非空判断。

对于某些特定需求，我们可以使用通配符进行匹配，比如：

```
../ * 用于查找所有同级Actor
/user/lookupActor/* 用于查找lookupActor的所有子级Actor
/user/lookupActor/target* 用于查找lookupActor的所有以target开头的子级Actor
```

除了这种方式外，还可以通过 ActorSelection 的 resolveOne 得到 ActorRef，该方法会返回一个 scala.concurrent.Future 对象（和 ask 方法返回的相同），当 Actor 不存在或者在指定的时间内没找到 Actor 时，会抛出 akka.actor.ActorNotFound 异常，示例代码如下：

```
ActorSelection as = system.actorSelection("/user/lookupActor/targetActor");
Timeout timeout = new Timeout(Duration.create(2, "seconds"));
Future<ActorRef> fu = as.resolveOne(timeout);
fu.onSuccess(new OnSuccess<ActorRef>() {
        @Override
        public void onSuccess(ActorRef ref) throws Throwable {
            System.out.println("查找到Actor: " + ref);
        }
}, system.dispatcher());

fu.onFailure(new OnFailure() {
@Override
        public void onFailure(Throwable ex) throws Throwable {
            if(ex instanceof ActorNotFound) {
                System.out.println("没找到Actor:"+ex.getMessage());
            }
        }
    }, system.dispatcher());
```

如代码所示，当查找到 Actor 时，可以通过 OnSuccess 回调得到 ActorRef 对象，而当查找不到时，通过 OnFailure 回调得到异常信息。

## 2.6　消息不可变

Actor 的所有任务都是消息驱动的，所以消息的正确性和有效性非常重要。理论上，Actor 可以传递或接收任何类型的消息，但是为了避免出现竞态条件（对共享资源的并发写）和不可预知的数据篡改，最好把消息设计成不可变的对象。在该对象中，我们通常会把所有变量设计成只读，在该对象的整个生命周期内，这些数据将不可修改。由于在 Java 中，并没有强制设置不变状态的手段，所以我们必须借助 final 关键字来实现，参考代码如下：

```
public final class Message {
    private final int visitCount ;
    private final List<String> usernames ;
    private final Map<String,String> citys ;
    public Message( int visitCount , List<String> usernames , Map<String, String>
citys) {
        this .visitCount = visitCount ;
        this .usernames = Collections.unmodifiableList( usernames);
        this .citys = Collections.unmodifiableMap (citys );
    }
    public int getVisitCount() {
        return visitCount ;
    }
    public List<String> getUsernames() {
        return usernames ;
    }
     public Map<String, String> getCitys() {
        return citys ;
    }
}
```

对于集合框架，有必要通过 Collections.unmodifiable 系列的 API 来保证不变性。Collections.unmodifiable 内部使用 UnmodifiableList、UnmodifiableMap 等 API 对现有的集合进行包装，它们通过重写其中的 set、add、put 等方法来达到限制数据修改的目的。

## 2.7　Actor 行为切换

在业务处理过程中，往往会面临一些关键数据的变化，而这些变化也会进一步影响着程序的执行逻辑，很多时候，我们把这些关键数据称之为“状态”。状态的处理需要依赖于具体的业务，而业务本身也可能充满不确定性，一种比较好的做法是，我们把多个状态的处理过程（即行为）封装起来，形成一个个独立的组件，在使用时就可以很方便地组装、重

用或切换。在 Actor 中，行为可以通过 akka.japi.Procedure 来定义，一旦抽取完我们所需要的 Procedure，就可以使用 become/unbecome 方法来切换他们，其中 become 表示切换为某个行为，unbecome 表示修改回上一个行为。

下面我们先通过一个小例子来演示一下这个过程。

首先创建一个普通的 Actor，取名 BecomeActor。在该 Actor 中，为了能实现切换的动作，需要先定义一个 Procedure 实例，并重写 apply 方法，具体代码如下：

```
class BecomeActor extends UntypedActor{
    Procedure<Object> proce= new Procedure<Object>() {
    @Override
    public void apply(Object message) {
        System. out .println("become: " +message );
        }
    };

    @Override
    public void onReceive(Object msg ) throws Exception {
    System. out .println("接收消息: " +msg );
    getContext().become( proce);
    System. out .println("-------------------------" );
    }
}
```

当 BecomeActor 接收到消息后，通过 getContext().becom() 方法切换到事先定义好的处理过程（Procedure），下面是测试代码：

```
ActorSystem system =ActorSystem. create( "sys" );
ActorRef ref =system .actorOf(Props.create (BecomeActor. class), "becomeActor");
ref.tell( "hello" ,ActorRef. noSender());
ref.tell( "hi" , ActorRef. noSender());
ref.tell( "hi" , ActorRef. noSender());
ref.tell( "hi" , ActorRef. noSender());
```

在测试代码中，我们向 BecomeActor 发送了 4 个消息，那么到底哪个消息会进入切换后的过程呢，还是先看看测试结果吧：

```
接收消息:hello
-------------------------
become: hi
become: hi
become: hi
```

我们会发现，当 BecomeActor 接收到一个消息并执行了 become 后，后面的所有消息都会进入新的 Procedure，感觉像是逻辑被完全替换一样，那么，它还能回头么？

回答是：当然可以。实际上，当调用一次 become 后，新 Procedure 的代码逻辑会被保存进一个执行栈中，此时可以通过 unbecome 来返回到上一个 Procedure，这里要注意的是：

这个过程还与 become 的第二个参数有关，该参数表示是否该存储当前行为，下面我们用一个更加完整的例子来说明这些细节问题。

在编写该示例之前，我们先想象一下这个虚拟场景：某公司在年尾时要计算每个员工的年终奖金，奖金的计算公式是：员工薪水 * 奖金比率，其中奖金比率根据级别的不同而不同，暂时我们只设定两个级别。在计算开始时，需要传入某一批员工的岗位级别，然后执行这一批员工的奖金计算，当计算执行到另外一个岗位级别时，需要把逻辑转入另一批员工的奖金计算流程。

为了描述每个员工的个人信息，比如姓名和薪水，首先需要一个 Emp 的实体类（该类就是普通的 entity，代码略），然后创建具体的业务 Actor，并在该 Actor 里定义两个 Procedure 分别来处理两个级别的奖金计算，具体代码如下：

```
class SimpleActorDemo extends UntypedActor{

    Procedure<Object> LEVEL1 = new Procedure<Object>() {
    @Override
    public void apply(Object message) {
    if (message instanceof String) {
    if (message .equals("end" )) {
    getContext().unbecome();
    }
    } else {
    Emp emp=(Emp) message;
    double result =emp .getSalary()*1.8;
    System. out .println("员工" +emp .getName()+ "的奖金为:  "+ result);
    }
    }
};

    Procedure<Object> LEVEL2 = new Procedure<Object>() {
    @Override
    public void apply(Object message) {
    if (message instanceof String) {
    if (message .equals("end" )) {
    getContext().unbecome();
    }
    } else {
    Emp emp=(Emp) message;
    double result =emp .getSalary()*1.5;
    System. out .println("员工" +emp .getName()+ "的奖金为:  "+ result);
    }
    }
};

    @Override
    public void onReceive(Object msg ) throws Exception {
    String level=(String) msg;
    if (level .equals("1" )) {
```

仅供非商业用途或交流学习使用

```
    getContext().become( LEVEL1);
    } else if (level .equals("2" )) {
    getContext().become( LEVEL2);
    }
}}
```

在 receive 方法里，首先判断传入的级别标志，然后切换到不同的处理过程。在 LEVEL1 和 LEVEL2 里，处理过程大致相似，主要是通过传入的员工信息和奖金比例来计算年终奖，当它们接收到 end 指令时，都会从当前行为切换到原来的 onReceive 里面，下面是测试代码：

```
ActorSystem system=ActorSystem. create( "sys");
ActorRef ref =system .actorOf(Props.create (SimpleActorDemo. class),
"simpleActorDemo");
ref.tell( "1",ActorRef. noSender());
ref.tell( new Emp("张三" , 10000),ActorRef.noSender());
ref.tell( new Emp("李四" , 20000),ActorRef.noSender());
ref.tell( "end" ,ActorRef.noSender());
ref.tell( "2" ,ActorRef.noSender());
ref.tell( new Emp("王五" , 10000),ActorRef.noSender());
ref.tell( new Emp("赵六" , 20000),ActorRef.noSender());
```

测试结果如下：

```
员工张三的奖金为: 18000.0
员工李四的奖金为: 36000.0
员工王五的奖金为: 15000.0
员工赵六的奖金为: 30000.0
```

通过结果可以看出，当 LEVEL1 接收到 end 指令并执行 getContext().unbecome() 方法后，计算流程会重新进入 onReceive 里。假如此时需要新加一个处理逻辑也比较简单，直接定义一个 Procedure，并且在合适的条件下挂载这个过程就可以了。现在，我们对刚才的代码做一些改造，首先新建一个 Procedure，取名 LEVEL3（它的代码和 LEVEL1、LEVEL2 类似，这里不再列出），然后在 LEVEL1 里加上 become LEVEL3 的代码，并传入 false，表示保留当前逻辑过程：

```
if (message .equals("become3" )) {
    getContext().become( LEVEL3, false );
}
```

并把之前测试代码中的第 6 行换成如下代码：

```
ref .tell("become3" ,ActorRef.noSender());
ref.tell( "end" ,ActorRef.noSender());
```

运行后的结果是：

```
员工张三的奖金为: 18000.0
员工李四的奖金为: 36000.0
```

仅供非商业用途或交流学习使用

```
员工王五的奖金为：18000.0
员工赵六的奖金为：36000.0
```

我们会发现，此时 unbecome 并没有将逻辑切换到 onReceive 里面，而是仍然回到 LEVEL1 这个 Procedure 去执行了。这主要是因为在调用 getContext().become（LEVEL3, false）时设置了第二个参数为 false，表示会在栈中保留当前 Procedure，当 unbecome 时，实际上是从 LEVEL3 回到了 LEVEL1，最后有一点要注意，当 Actor 由于某种原因导致重启后，将恢复成最初的行为。

## 2.8　Actor 生命周期

Actor 在运行时中会经历不同的阶段，比如创建、运行、重启和销毁等，这一系列的过程或状态，我们称之为生命周期。在理想情况下，Actor 应该像一头老黄牛一样任劳任怨地不停工作，但实际情况往往是，当某个不算常态的错误发生时（比如网络偶尔超时），我们想让它重启一下，然后重复之前的动作；又或者，当一个程序异常被抛出时，我们不得不停止该 Actor，让它不能继续走下去，这些行为有可能是手动触发的，也可能是 Actor 的监督 - 容错机制导致的（具体的容错处理将在 2.10 节详细介绍）。无论怎样，当这些行为发生时，我们都希望能及时感知并做有效处理。

下面我们首先通过一张图来看看 Actor 生命周期的具体内容（图 2-10）。

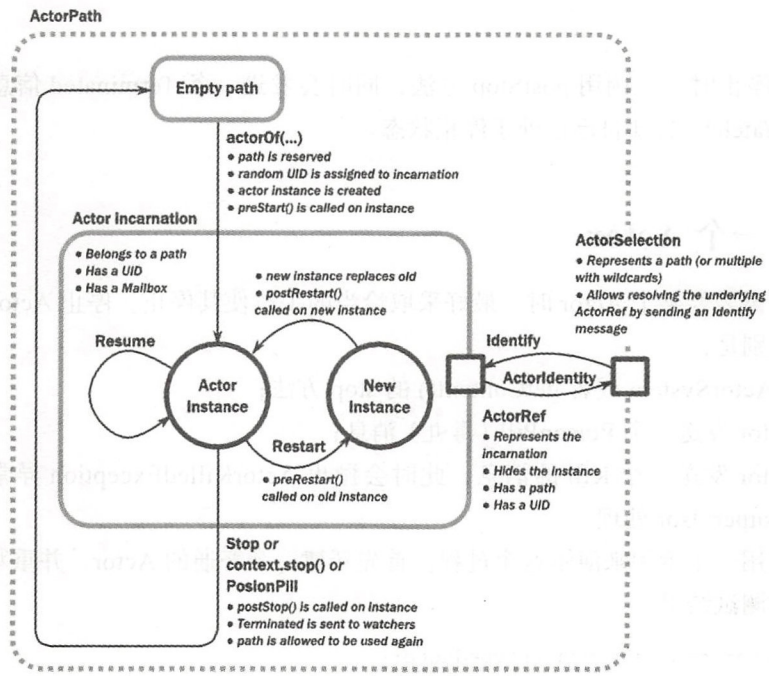



图 2-10　Actor 生命周期（摘自官网）

仅供非商业用途或交流学习使用

从图 2-10 中可以看到，Actor 生命周期主要包括启动（Start）、恢复（Resume）、重启（Restart）、停止（Stop）这四个阶段，每个阶段都会伴随着自身状态信息的变化。

**创建并启动**

当通过 actorOf 创建并启动 Actor 时，该 Actor 除了默认拥有已指定的 path 外，也会被分配一个 UID（可以通过 getSelf().path().uid() 得到该值），作为它的唯一标识。在 Actor 启动后，会默认调用 preStart 方法，在该方法里，我们可以做些资源初始化的操作。

**恢复运行**

当 Actor 出现某种异常后，通过容错机制，可以让该 Actor 恢复并继续运行，此时 Actor 会延用之前的实例，状态也会被保留下来。

**重启**

当 Actor 出现某种异常后，通过容错机制，可以让该 Actor 执行重启。重启的具体过程如下：

1）调用旧实例的 preRestart 方法，该方法默认情况下会停掉所有子级 Actor，并调用 postStop 方法；

2）创建新实例，并在新实例上调用 postRestart 方法，该方法默认情况下会调用 preStart 方法。

Actor 重启后，path 和 UID 不变，这意味着假如要继续使用该 Actor，不需要重新获取 ActorRef 对象，使用之前那个就可以了。同时也得注意到：Actor 重启后不会保留自身状态。

**停止**

当 Actor 停止时，会调用 postStop 方法，同时会发送一条 Terminated 信息给自己的监控者（DeathWatch），告知自己已处于停止状态。

## 2.9 停掉一个 Actor

当我们不再需要某个 Actor 时，最好采取恰当的手段使其停止。停止 Actor 大致有三种方式，它们分别是：

- 调用 ActorSystem 或者 getContext() 的 stop 方法；
- 给 Actor 发送一个 PoisonPill（毒丸）消息；
- 给 Actor 发送一个 Kill 的消息，此时会抛出 ActorKilledException 异常，并上报到父级 supervisor 处理。

下面我们用一个示例来演示这个过程。首先新建一个普通的 Actor，并重写 postStop 方法，以便观察测试结果：

```
class WorkerActor extends UntypedActor{
    LoggingAdapter log =Logging.getLogger( this .getContext().system(), this);
```

仅供非商业用途或交流学习使用

```
@Override
    public void onReceive(Object msg ) throws Exception {
        log.info( "收到消息: " +msg );
    }

    @Override
    public void postStop() throws Exception {
        log.info( "Worker postStop" );
    }
}
```

测试代码如下：

```
ActorSystem system=ActorSystem. create( "sys");
ActorRef ar =system .actorOf(Props.create (WorkerActor. class), "workerActor");
system.stop(ar);
```

此时会打印“Worker postStop”，表示 Actor 已经关闭，这里我们也可以把代码改为

```
ar.tell(PoisonPill.getInstance(),ActorRef.noSender())
```

或者

```
ar .tell(Kill.getInstance(),ActorRef. noSender());
```

都可以实现类似功能，同时我们也会注意到，在使用 kill 方式的时候会多出一个 Error 的日志：

```
[ERROR] [07/24/2016 23:57:00.021] [sys-akka.actor.default-dispatcher-2] [akka://
    sys/user/workerActor]
Kill (akka.actor.ActorKilledException)
```

这是因为，在通过 kill 方式杀死 Actor 时会抛出 akka.actor.ActorKilledException 异常，该异常会被父级 Supervisor 处理，而默认的处理方式就是停止该 Actor。

通过上面的代码我们会发现，停止一个 Actor 是如此的简单，仅仅需要一句代码就可以轻松搞定，但是随意停掉一个 Actor 往往也会带来不可预知的后果。假如要实现一个更加复杂的应用，我们还需要探究该行为背后的整个过程，实际上，为了提高可用性和维护性，Actor 在停止时都会遵循一套比较可靠的流程：

1）当停止 Actor 时，正在处理的消息会在完全停止之前处理完毕，后续信息将不再进行处理，邮箱（用来保存 Actor 的消息）将被挂起；

2）给所有子级 Actor 发送终止指令，当子级都停掉后，再停掉自己，停止完毕后会调用 postStop 方法，在这里可以清理或释放资源；

3）向生命周期监控者（DeathWatch）发送 Terminated 消息，以便监控者做相应的处理。

现在我们再编写一个 WatchActor 作为父级来创建上面的 WorkerActor，继续测试一下上面所说的整个流程，WatchActor 代码如下：

仅供非商业用途或交流学习使用

```
class WatchActor extends UntypedActor{

LoggingAdapter log =Logging.getLogger( this .getContext().system(), this);
    ActorRef child= null ;

    @Override
    public void preStart() throws Exception {
        //创建子级Actor
        child=getContext().actorOf(Props. create(WorkerActor. class), "workerActor" );
    }
    @Override
    public void postStop() throws Exception {
        log.info( "WatchActor postStop" );
    }
    @Override
    public void onReceive(Object msg ) throws Exception {
    }
}
```

当创建 WatchActor 时，会默认启动 preStart 方法，该方法中创建了一个子级 Actor，此时停止 WatchActor 时，会有如下结果：

```
[INFO] [07/25/2016 16:33:33.552] [sys-akka.actor.default-dispatcher-7]
[akka://sys/user/watchActor/workerActor]Worker postStop
[INFO] [07/25/2016 16:33:33.552] [sys-akka.actor.default-dispatcher-6]
[akka://sys/user/watchActor] WatchActor postStop
```

该结果说明，在停止层级 Actor 时，会先停掉子级，再停掉父级。

为了监控到子级 Actor 的停止操作，我们需要调用 getContext().watch 方法对其进行监听，当 Actor 真正停掉后，监控者就可以接收到 Terminated 消息。这里我们将 WatchActor 进行一些改造，具体代码如下：

```
class WatchActor extends UntypedActor{
        LoggingAdapter log =Logging.getLogger( this .getContext().system(), this);
        ActorRef child= null ;

    @Override
    public void preStart() throws Exception {
            //创建子级Actor
            child=getContext().actorOf(Props. create(WorkerActor. class), "workerActor"
                );
            //监控child
            getContext().watch( child);
    }

    @Override
    public void postStop() throws Exception {
        log.info( "WatchActor postStop" );
    }
```

仅供非商业用途或交流学习使用

```
    @Override
    public void onReceive(Object msg ) throws Exception {
    if (msg instanceof String) {
        if (msg .equals("stopChild" )) {
            getContext().stop( child);
        }
    } else if (msg instanceof Terminated) {
            Terminated t=(Terminated) msg;
            log.info( "监控到" +t .getActor()+ "停止了");
    } else {
            unhandled( msg);
    }
  }
}
```

此时给 WatchActor 发送 stopChild，会得到如下结果：

```
[INFO] [07/25/2016 16:39:44.817] [sys-akka.actor.default-dispatcher-7] [akka://
    sys/user/watchActor/workerActor] Worker postStop
[INFO] [07/25/2016 16:39:44.821] [sys-akka.actor.default-dispatcher-6]
[akka://sys/user/watchActor] 监控到Actor[akka://sys/user/watchActor/workerActor#-
    1416122738]停止了
```

如结果所示，当 workerActor 停止后，watchActor 收到了它停止的消息。当不再需要该监控时，可以调用 getContext().unwatch(child) 进行解除。

## 2.10 监督与容错处理

对于一个大型的应用程序来讲，由于服务器、网络、数据、程序代码都可能会出现问题，所以笔者比较倾向于认为："程序失败" 100% 会发生，在开发阶段，我们必须要有 "治未病" 的思想。当失败发生时，我们要尽可能保证对用户的服务不受影响，这就要求该系统必须拥有良好的容错机制和高可用保障，能够在服务意外崩溃或不可用时进行自我修复。

在 Akka 中，Actor 作为最基本的执行单元，承载着所有业务的核心，在消息处理时产生的任何失败都会对整个应用造成伤害，所以必须对它做一些容错处理。在之前 2.12 节中我们提到过，Actor 系统采用 "父监督" 的模式进行管理，即父 Actor 会监督子 Actor 的异常情况，然后根据默认或者预设的处理逻辑来确定到底是该恢复 Actor、停止 Actor、重启 Actor 还是把错误上溯到父级。

Akka 提供了两种监督策略，分别是 One-For-One Strategy 和 All-For-One Strategy，前者表示当一个子 Actor 出现异常时，只对该 Actor 做处理，后者则表示对所有子 Actor 都做处理，大部分时候应该选择 One-For-One Strategy（这也是默认的策略），除非子 Actor 之间的业务有很强的关联或者互相依赖。当程序中没有显式指定策略时，会启动一个默认策略，该策略遵循下列规则（如图 2-11 所示）：

仅供非商业用途或交流学习使用

- 当抛出 ActorInitializationException 和 ActorKilledException 时，会终止子 Actor；
- 当抛出 Exception 时，会重启子 Actor；
- 抛出其他类型的 Throwable 异常时，会上溯失败到父级。

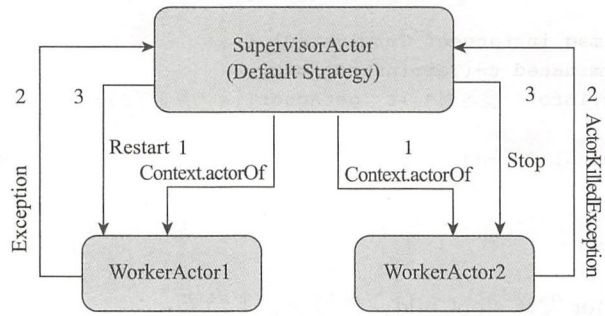



图 2-11 Actor 默认监督策略

在 Actor 执行任务时可能会抛出不同类型的异常，很多时候，我们应该知道，哪些异常需要让 Actor 停止或重启，哪些异常可以“视而不见”。为了更加细化这种判断，我们需要自定义监督逻辑。

自定义一个监督逻辑非常简单，只需要在父 Actor 中创建一个 SupervisorStrategy 对象，并通过 supervisorStrategy() 方法返回出来，示例代码如下：

```
public class SupervisorActor extends UntypedActor {
    //定义监督策略
    private SupervisorStrategy strategy = new OneForOneStrategy(3,
    Duration.create("1 minute"), new Function<Throwable, Directive>() {

@Override
    public Directive apply(Throwable t) {
if (t instanceof IOException) {
            System.out.println("==========IOException=========");
            return SupervisorStrategy.resume();
    } else if (t instanceof IndexOutOfBoundsException) {
    System.out.println("=========IndexOutOfBoundsException==========");
        return SupervisorStrategy.restart();
    } else if (t instanceof SQLException) {
    System.out.println("==========SQLException=========");
        return SupervisorStrategy.stop();
    } else {
            System.out.println("==========escalate=========");
            return SupervisorStrategy.escalate();
}
    }
});

    //返回监督策略，使其可用
```

仅供非商业用途或交流学习使用

```
    @Override
    public SupervisorStrategy supervisorStrategy() {
    return strategy;
    }

    @Override
    public void onReceive(Object msg) throws Exception {
    }
}
```

在 SupervisorActor 中，定义了一个 OneForOneStrategy 对象，该对象需要三个参数，分别是 maxNrOfRetries、withinTimeRange、decider。前两个表示在指定时间内的最大重启次数，超过这个次数就 stop 掉，在我们的代码中设定了：在 1 分钟内重启超过 3 次则停止该 Actor。第三个参数 decider 是一个 Function 对象，它通过 apply 方法返回监督指令（Directive），这些指令有：

- 恢复运行，API 是 SupervisorStrategy.resume()；
- 重启，API 是 SupervisorStrategy.restart()；
- 停止，API 是 SupervisorStrategy.stop()；
- 升级失败，API 是 SupervisorStrategy.escalate()。

为了验证这些指令是否正常工作，我们还需要建立一个子 Actor，并且做两个重要工作：

1）重写 preStart、postStop，preRestart，postRestart 这几个方法，观察不同情况下的生命周期状况；

2）定义一个状态数据，观察它在不同情况下的变化。

下面是子 Actor 的示例代码：

```
class WorkerActor extends UntypedActor {
    //状态数据
    private int stateCount=1;
    public void preStart() throws Exception {
        super.preStart();
        System.out.println("worker actor preStart");
    }
    @Override
    public void postStop() throws Exception {
        super.postStop();
        System.out.println("worker actor postStop");
    }
    @Override
    public void preRestart(Throwable reason, Option<Object> message)
            throws Exception {
        System.out.println("worker actor preRestart begin "+this.stateCount);
        super.preRestart(reason, message);
        System.out.println("worker actor preRestart end " +this.stateCount);
    }
```

仅供非商业用途或交流学习使用

```
    @Override
    public void postRestart(Throwable reason) throws Exception {
        System.out.println("worker actor postRestart begin " +this.stateCount);
        super.postRestart(reason);
        System.out.println("worker actor postRestart end " +this.stateCount);
    }
    @Override
    public void onReceive(Object msg) throws Exception {
        //模拟计算任务
        this.stateCount++;
        if (msg instanceof Exception) {
            throw (Exception)msg;
        }else if("getValue".equals(msg)){
            //返回当前数据
            getSender().tell(stateCount,getSelf());
        }else {
            unhandled(msg);
        }
    }
}
```

下面我们来看看，给 WorkerActor 发送不同类型的异常后，它的生命周期和状态到底会发生哪些变化。

首先，在 SupervisorActor 中的合适位置（比如 preStart 方法里）创建子 WorkerActor 并监控它：

```
//创建子Actor
ActorRef workerActor =getContext().actorOf(Props.create(WorkerActor.class),"worderActor");
//监控生命周期
getContext().watch(workerActor);
```

然后在 SupervisorActor 接收返回的状态数据和子 Actor 的终止消息：

```
if(msg instanceof Terminated) {
            Terminated ter=(Terminated)msg;
            System.out.println(ter.getActor()+"已经终止");
        }else {
            System.out.println("stateCount="+msg);
        }
```

测试过程也比较简单，只需要发送异常信息和得到状态数据就可以了：

```
workerActor.tell(new IOException(), getSelf());
//workerActor.tell(new SQLException("SQL异常"),getSelf());
//workerActor.tell(new IndexOutOfBoundsException(), getSelf());
  workerActor.tell("getValue",getSelf());
```

上面主要测试了 resume、stop、restart 三个指令的功能，具体日志内容这里不再贴出。表 2-1 是该测试的结果及相关说明。

仅供非商业用途或交流学习使用

表 2-1　Supervisor 测试结果说明

| 异　常 | 运行结果及说明 |
| --- | --- |
| IOException | WorkerActor 匹配到 resume 指令，然后恢复继续运行。状态 stateCount=3，说明该过程可以保留之前的状态。 |
| SQLException | WorkerActor 匹配到 stop 指令然后停止，后续消息将接收不到。SupervisorActor 接收到了 WorkerActor 的 Terminated 消息。 |
| IndexOutOfBoundsException | WorkerActor 匹配到 restart 指令，然后进行重启。重启时，会先在旧实例上调用 preRestart 方法，该方法默认会停止自己的所有子级并调用 postStop 方法，然后在新实例上调用 postRestart 方法，该方法默认会调用 prestart 方法。重启后 stateCount=2，说明该过程会清空之前的状态。 |

## 2.11　Circuit Breaker（熔断）

在分布式环境中往往会出现各种各样的问题，比如网络异常、超时等，而这些问题可能会导致系统的级联失败，如果不断重试的话，可能也是无法解决的，还可能耗费更多的资源。

举个例子，假如我们有一个 Web 程序，它与第三方的 Web 服务相关联，这个第三方由于某些负载原因导致数据库挂掉了，超时了。对于我们的系统来讲，用户会被长时间 hang 住，然后用户可能会不断刷新，做更多的请求。这最终会让 Web 程序由于资源耗尽而导致失败。而 circuit breaker 可以让请求快速失败，并给用户反馈错误信息，这样 circuit breaker 就可以用来防止应用不断尝试一个可能失败的操作。

如图 2-12 所示，circuit breaker 有三个状态，分别是：Closed、Open、Half-Open。

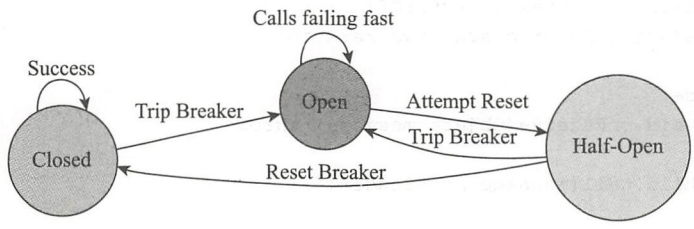



图 2-12　Circuit Breaker 状态图（图片来自 Akka 官网）

**Closed 状态**

1）正常情况下，circuit breaker 是 closed 状态：异常或者调用超过配置的 callTimeout，就增加一次失败计数；

2）成功则重置失败计数为 0；

3）当失败次数到达 maxFailures，会进入 Open 状态。

**Open 状态**

1）所有调用伴随着抛出 CircuitBreakerOpenException，立即失败；

2）在 resetTimeout 后，circuit breaker 进入 Half-Open 状态。

Half-Open 状态

1）第一次调用尝试运行进入而不会快速失败；

2）所有其他调用会处于 Open 状态，进而快速失败；

3）如果第一个调用成功，会重回 Closed 状态；

4）如果第一个调用失败，会进入 Open 状态，然后等待下一次 resetTimeout。

Akka 的 akka.pattern.CircuitBreaker 提供了对 circuit breaker 的直接支持。当创建 CircuitBreaker 时，我们需要传入 maxFailures、callTimeout、resetTimeout，下面是一个简单示例：

```
public class CircuitBreakerActor extends UntypedActor{
    private ActorRef workerChild;
    private static SupervisorStrategy strategy = new
OneForOneStrategy(20,
    Duration.create("1 minute"), new Function<Throwable,
Directive>() {
    @Override
    public Directive apply(Throwable t) {
    return SupervisorStrategy.resume();
    }});
    @Override
    public SupervisorStrategy supervisorStrategy() {
    return strategy;
    }
    @Override
    public void preStart() throws Exception {
    super.preStart();
    workerChild=getContext().actorOf(
Props.create(WorkerActor.class),"workerActor");
    }
    @Override
    public void onReceive(Object message) throws
Exception {
    workerChild.tell(message, getSender());
    }
}
class WorkerActor extends UntypedActor{
    private CircuitBreaker breaker;
    @Override
    public void preStart() throws Exception {
    super.preStart();
    this.breaker = new
CircuitBreaker(getContext().dispatcher(),
    getContext().system().scheduler(), 5,
Duration.create(2, "s"),
    Duration.create(1, "min")).onOpen(new Runnable() {
    public void run() {
```

```
        System.out.println(DateUtils.getCurrentDateStr()+"-
    -->Actor CircuitBreaker 开启");
        }
        }).onHalfOpen(new Runnable() {
        @Override
        public void run() {
        System.out.println(DateUtils.getCurrentDateStr()+"-
    -->Actor CircuitBreaker 半开启");
        }
        }).onClose(new Runnable() {
        @Override
        public void run() {
        System.out.println(DateUtils.getCurrentDateStr()+"--->Actor CircuitBreaker关闭");
        }
        });
    }
        @Override
        public void onReceive(Object message) throws Exception {
        if (message instanceof String) {
        String msg = (String) message;
        if (msg.startsWith("sync")) {
        getSender().tell(breaker.callWithSyncCircuitBreaker(new Callable<String>() {
            @Override
            public String call() throws Exception {
                System.out.println("msg:" + msg);
                Thread.sleep(3000);
                return msg;
            }
        }), getSelf());
        }}}}
```

为了更好地演示效果，我们设置了 SupervisorStrategy.resume() 的策略，让 Actor 在出现异常或超时的时候能继续运行。在 WorkerActor 中，我们在启动阶段创建了 CircuitBreaker 对象，并且设置了 maxFailures=5、callTimeout=2 秒、resetTimeout=1 分钟。当向 WorkerActor 转发“sync”为开头的字符串后，会进入同步处理的代码里，在这里，我们等待了 3 秒钟，以便能触发超时并计数一次，当计数达到 5 次后，CircuitBreaker 会处于 Open 状态，此时回调函数 onOpen 会被调用，而后续的处理会直接失败（calls are failing fast）。1 分钟（resetTimeout）之后，circuit breaker 进入 Half-Open 状态，此时回调函数 onHalfOpen 会被调用。继续发送消息后，假如被成功处理，那么 circuit breaker 会进入 Closed 状态，并调用 onClose 函数，假如失败，则进入 Open 状态。

这段代码演示的是同步调用方式，大家也可以尝试一下它的异步 API。

## 2.12　配置相关

Akka 程序在启动时会加载一些默认的配置项，所以我们不需要显式地提供配置文件，

仅供非商业用途或交流学习使用





假如要个性化配置某些行为，则需要新建配置文件。在默认情况下，程序会加载 classpath 下的 application.conf、application.json、application.properties 文件，对于比较简单的程序，可以直接配置 application.conf：

```
akka {
    log- config-on-start = on
}
```

或者

```
akka.log- config-on-start = on
```

这里把 log-config-on-start 配置成 on，表示启动时（即调用 ActorSystem. create() 方法后）打印配置信息。

**多配置文件**

对于比较大的应用来讲，可能一个配置文件并不能覆盖所有的情况，常规做法是把配置文件拆分成多个，然后通过 include 的方式进行引入，比如下面这个名为 appsys.conf 的配置文件：

```
include "application"
akka {
    loglevel = "DEBUG"
}
```

它引入了 application.conf 文件，然后增加了 loglevel 的配置。在创建 ActorSystem 时，可以显式加载 appsys.conf，代码如下：

```
ActorSystem system=ActorSystem. create( "mysys",ConfigFactory.
load( "appsys" ));
```

Akka 提供的配置项涵盖了几乎所有的核心功能，比如日志、远程、路由、消息序列化、分布式集群等，这里笔者只是对它做了初步的介绍，更多用法将在后续章节中进行详细讲解。

## 2.13 本章小结

本章首先介绍了 Actor 组件相关的概念，然后详细讲解了 Actor 的基本用法，以及怎样实现一个具有容错能力的 Actor 系统，除此之外，大家也应该了解如下几点：

- Actor 运行于线程池之上，提供了比线程更加轻量级的并行运行方案，并且每个 Actor 都是线程安全的，不必考虑同步或锁的问题，同时要注意，为了保证消息的状态不受污染，最好采用不变类型的对象；
- Actor 采用“问答式”的 API 实现消息的异步通信，并且在内部串行处理接收到的

仅供非商业用途或交流学习使用



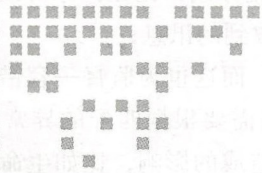


消息，这非常贴近真实社会中人与人之间的交流方式，比如，你可以和多个人进行交谈，但是需要自己一条条顺序地去“消化”收到的讯息；

- 在一个 Actor 系统中，通常会形成层级的网络，而这也为监督—容错功能提供了基础。实现一个自定义的监督策略非常简单，只需要根据匹配的异常来指定不同的指令，但是我们必须了解不同指令对子 Actor 造成的影响，比如生命周期和状态的变化。

仅供非商业用途或交流学习使用





Chapter 3 第 3 章

# 线程调度

在 Akka 中，Actor 的消息通信和任务执行建立在一个完全透明的调度机制之上，它屏蔽了底层线程（池）的实现细节，几乎让开发者感觉不到它的存在。调度机制类似于一个消息中心控制器，所有消息、任务都由它负责分发和派送，在这个过程中，有很多因素会影响整个系统的效率和性能。在本章中，笔者将介绍线程调度（dispatcher）的基本概念及用法。

## 3.1 什么是 Dispatcher ?

在解释什么是 Dispatcher 之前，我们先看一个现实生活中的真实场景：当你在某电商网站上购买一件商品后，在一般情况下，快递小哥会按时给你送货，但是在“双十一”时，由于每个快递小哥都有大量的货要送，所以很可能会造成延迟送货。为了解决这个问题，快递公司可能会从两方面入手：

1）让快递小哥们提高单次送货速度，尽可能多送货；

2）临时增派人手，以便处理更多的送货任务。

实际上，这两个方案都有其局限性。对于第一个方案，有时候单次速度并不会提高得太明显，作用有限；对于第二个方案，当“双十一”的购买洪峰褪去后，这些增派的人手会被闲置，造成成本上升。

这个场景其实和 Actor 的消息处理非常类似。在 Akka 中，Actor 运行在一个线程池之上，每个线程（快递小哥）会处理多个 Actor 消息（商品），为了能保证 Actor 消息处理的及时性和线程的使用效率，我们需要对线程池做一些调配或协调的工作，这就是 Dispatcher 的意义所在（如图 3-1 所示）。

仅供非商业用途或交流学习使用





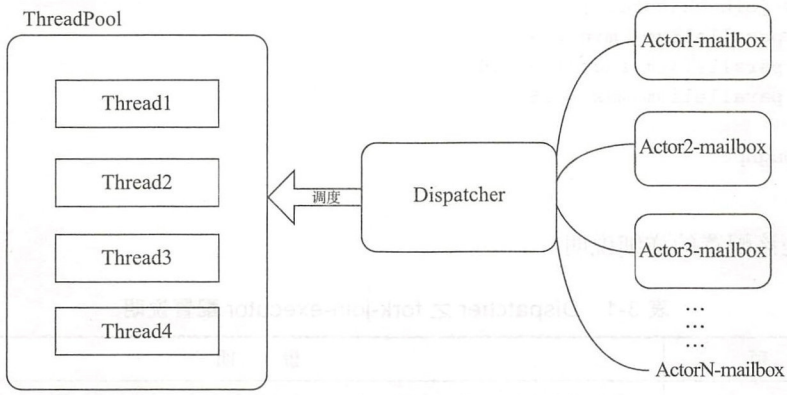



图 3-1 Dispatcher 线程调度

在 Akka 中，Actor 的任务（消息）由 Dispatcher 进行分配并执行，所以它基本上是整个系统吞吐量的最关键因素。在实际项目中，我们（Actor）可能需要处理各种不同类型的任务，有的任务耗时长，或者长时间阻塞，而有的任务则处理较快，不会造成阻塞，大部分时候，我们会将这两类任务做线程资源上的隔离，即分配不同的 Dispatcher 处理它们，此时可以避免由于某类任务耗尽线程而让其他请求等待的情况。

## 3.2 Executor 选择

Executor 为 Dispatcher 提供了执行异步任务的策略，它的实现方式有如下两种：

- thread-pool-executor：基于普通的线程池，它有一个工作队列，当线程空闲时会从队列中获取任务并执行。
- fork-join-executor：基于工作窃取的线程池，它的工作思路是把大的计算任务拆分成小的任务然后并行执行，最后合并结果。当某个线程的任务队列里没有任务时，会主动从其他线程的任务队列中“窃取”任务。一般认为，fork-join 的性能更佳，这也是 Akka 的默认选项。

## 3.3 配置 Dispatcher

每个 ActorSystem 都有一个默认的 Dispatcher 配置项，并且使用 fork-join 作为默认的 Executor 配置，假如需要自定义 Dispatcher，则可以进行如下配置：

```
my-forkjoin-dispatcher {
type = Dispatcher
executor = "fork-join-executor"
    # 配置fork-join线程池
```

仅供非商业用途或交流学习使用





```
    fork-join-executor {
        parallelism- min = 3
        parallelism-factor = 3.0
        parallelism-max = 15
    }
    throughput = 1
}
```

表 3-1 是该配置的详细说明。

表 3-1 Dispatcher 之 fork-join-executor 配置说明

| 配 置 项 | 说 明 |
| --- | --- |
| my-forkjoin-dispatcher | 自定义的 Dispatcher 名字，Actor 通过该名字来进行 Dispatcher 配置 |
| type | Dispatcher 的类型，除了 Dispatcher 之外，还支持 PinnedDispatcher、CallingThreadDispatcher，不同的 type 有不同的线程调度策略 |
| parallelism- min | 最小并发线程数 |
| parallelism-factor | 并发因子，该值会影响线程池的最大可用线程数，具体的计算方式是：最大线程数 = 处理器个数 *parallelism-factor，比如当服务器 CPU 处理器个数是 4，并发因子是 3，那么最大线程数就是 12，在 Java 中，可以通过 Runtime. getRuntime(). availableProcessors() 得到处理器个数 |
| parallelism-max | 最大并发线程数，在程序运行过程中，实际的最大线程使用数量 =min（parallelism-max，处理器个数 *parallelism-factor），即 parallelism-max 和上面计算的最大线程数较小的那个 |
| throughput | 对于一个 Actor，某个线程在（返回到线程池）处理下一个 Actor 之前能处理的最大消息数，假如设置为 1，表示尽可能公平的处理消息 |

thread-pool-executor 的配置与 fork-join-executor 非常类似：

```
my- threadpool-dispatcher {
    # dispatcher类型
    type = Dispatcher
    executor = "thread-pool-executor"
    # 配置线程池
    thread-pool-executor {
    # 最小线程数
    core-pool-size- min = 2

    #  并发使用的最大线程数=处理器*因子
    core-pool-size-factor = 3.0

    #最大线程数
    core-pool-size-max = 15
    }
    throughput = 1
}
```

仅供非商业用途或交流学习使用





## 3.4 使用 Dispatcher

Actor 使用 Dispatcher 有两种方式：代码和配置。

代码示例如下：

```
ActorRef ref =system .actorOf(Props. create(ActorDemo. class)
.withDispatcher( "my-forkjoin-dispatcher"), "actorDemo");
```

即通过 withDispatcher 来指定 Dispatcher 的名字。

假如不想硬编码使用 Dispatcher，可以通过如下方式，给某个 Actor 配置一个 Dispatcher：

```
akka.actor.deployment {
    /actorDemo {
        dispatcher = my-forkjoin-dispatcher
    }
}
```

## 3.5 PinnedDispatcher

PinnedDispatcher 是另外一种 Dispatcher 类型，它会为每个 Actor 提供只有一个线程的线程池，该线程池为 Actor 独有。当 Actor 处理比较耗时、阻塞的操作时，使用 PinnedDispatcher 将会非常有效。

PinnedDispatcher 的配置如下：

```
my-pinned-dispatcher {
    executor = "thread-pool-executor"
    type = PinnedDispatcher
}
```

大部分时候，PinnedDispatcher 会和 thread-pool-executor 一起使用。

为了更好地理解 PinnedDispatcher 和 Dispatcher 这两种调度类型的区别，我们可以创建一些耗时的 Actor，并且批量地对这些 Actor 发起请求，然后观察处理过程的不同。

首先，定义一个 Actor，通过 Thread.sleep(5000) 来模拟耗时操作：

```
class ActorPinnedDemo extends UntypedActor{
    @Override
    public void onReceive(Object msg) throws Exception {
        System.out.println(getSelf()+"--->"+msg+""
+Thread.currentThread().getName());
            Thread.sleep(5000);
    }
}
```

然后批量创建 Actor，并给这些 Actor 发送信息：

仅供非商业用途或交流学习使用



```
for(int i=0;i<20;i++) {
    ActorRef ref = system.actorOf(Props.create(ActorPinnedDemo.class)
        .withDispatcher("my-pinned-dispatcher"),"actorDemo"+i);
    ref.tell("hello pinned",ActorRef.noSender());
}
```

运行该代码，会发现每个 Actor 将会很快地收到信息，而把 my-pinned-dispatcher 换成 my-forkjoin-dispatcher 或者 my-threadpool-dispatcher 后，会发现 Actor 在处理消息时有明显的等待现象（前提是请求量要大于 Dispatcher 设置的最大线程数量）。之所以会出现这种情况是因为：

- PinnedDispatcher 会让每个 Actor 都有自己单独的线程池（虽然只有一个线程），这个线程池不会被其他 Actor 共享；
- 而普通的 Dispatcher 会让多个 Actor 绑定到一个线程池，当任务很多并且非常耗时的时候，会占用大量（甚至全部）的线程去执行它们，而新的任务会等待空闲的线程。

虽然说 PinnedDispatcher 可以避免由于阻塞导致系统整体吞吐率的下降，但是过多地使用它也可能会导致服务器资源耗尽，在使用 Akka 的过程中，我们需要尽量避免阻塞的情况。

## 3.6 本章小结

Akka 提供了透明的调度机制来驱动 Actor 的运行，即使不显式地配置它们，系统也会默认采用适用场景最广泛的 Dispatcher 和性能最佳的 fork-join-executor 来进行线程调度。为了尽可能提高系统的性能和吞吐率，我们需要对某些关键属性进行配置，比如最小 / 最大线程数、并发因子等，而且这些取值都应该根据系统的整体情况来综合考虑。

仅供非商业用途或交流学习使用

第 4 章 Chapter 4

# 邮　箱

当我们在给一个 Actor 发送消息时，并不是直接依赖 API 之间的传参方式，而是会把消息先发送到 Actor 的邮箱（mailbox）里面，然后 Actor 会从自己的邮箱里取出这些消息，就好比一个快递员在给你送货时，会把物品直接放在你家门口的篮子里，这样你可以从篮子里面取出自己的物品。Actor 中的邮箱是一个队列结构，所有发送过来的消息都会在该队列进行排队，在默认情况下，它遵循先进先出（FIFO）的模式，假如需要改变这种默认处理方式，需要自定义邮箱或消息队列。

本章主要介绍了邮箱的相关配置以及不同邮箱类型之间的差异，然后演示如何自定义邮箱。

## 4.1　消息处理顺序

Actor 既可以存在于本地，也可以存在于远程的某个节点，并且它也可以和任何其他节点的 Actor 进行消息通信，有可能一个 Actor 会接收来自不同 Actor 的消息，此时该 Actor 并不能保证某个消息一定排在另一个的前面或后面（当然，也可以有办法解决），但是对于从同一个 Actor 发过来的多个消息，它能保证串行的顺序，如图 4-1 所示。

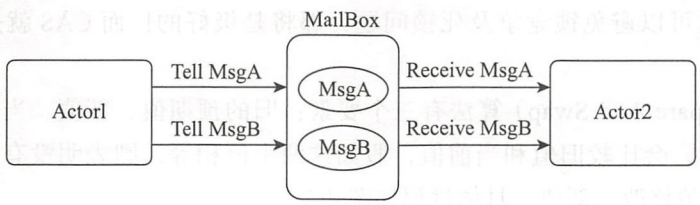



图 4-1　Actor 串行接收同一个 Actor 发送过来的多条信息

当 Actor1 给 Actor2 连续发送 MsgA 和 MsgB 后，Actor2 可以串行地处理 MsgA 和 MsgB 两个消息。

## 4.2 默认邮箱配置

Akka 对邮箱提供了专门的配置项，即默认邮箱配置（default-mailbox），比如邮箱类型（mailbox-type）、邮箱容量（mailbox-capacity）、入队超时时间（mailbox-push-timeout-time）等，它看起来是这样的：

```
akka.actor.default-mailbox {
        mailbox-type = "akka.dispatch.UnboundedMailbox"
        mailbox-capacity = 1000
        mailbox-push-timeout-time = 10s
    }
```

当我们不对邮箱做任何配置时，都会采用上面这个 default-mailbox 的默认配置。下面来看看这几个配置的具体含义。

- mailbox-type：邮箱类型，分为有界（Bounded）和无界（Unbounded），Akka 默认采用 UnboundedMailbox，表示不限制邮箱队列的大小；
- mailbox-capacity：邮箱容量，定义了有界邮箱（BoundedMail）的大小，该值只能是正数；
- mailbox-push-timeout-time：入队超时时间，主要是指 push 一个消息到有界邮箱的队列的超时时限。假如为负数，则表示无限超时，这可能会带来死锁问题。

UnboundedMailbox 主要依赖于 java.util.concurrent.ConcurrentLinkedQueue 来实现，从名字就可以看出来，它是一个基于链表的队列结构，并且遵循 FIFO（先进先出）。具体来讲就是，使用该队列时，新加的元素会被放在队列的尾部，而取元素时会从头部开始。另外，它通过 CAS 无锁算法来保证多线程下的安全，性能上有很大优势。

那么，CAS 又是什么呢？

我们在处理多线程程序的并发问题时，经常会使用锁机制来保证线程的安全，特别是对于公共的临界资源，宁可舍弃部分性能也要保证资源状态的安全，基本上可以认为，锁是造成系统性能及吞吐率下降的最重要原因之一。假如此时有一种无锁的算法，既能保证安全问题，又可以避免锁竞争及死锁问题，那将是极好的！而 CAS 就是这样一种无锁算法。

CAS（Compare And Swap）算法有三个要素：旧的预期值、新值、当前值，在对某个状态进行更新时，会比较旧值和当前值，假如这两个值相等，则表明没有被其他线程侵入过，此时可以把值修改为新值，具体过程如图 4-2 所示。

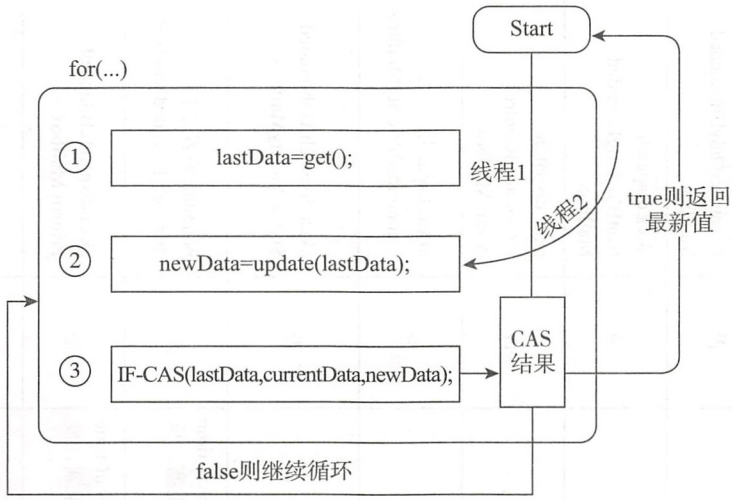



图 4-2 CAS 算法过程

## 4.3 内置邮箱

### 4.3.1 内置邮箱说明

除了默认的 UnboundedMailbox 邮箱之外，Akka 还提供其他多种类型的邮箱，以满足不同场景的需要。在 Akka 中，邮箱主要分为两大类：Unbounded 和 Bounded。Unbounded 表示无界，即邮箱没有容量上的限制；Bounded 表示有界，即邮箱有容量上的限制。大部分无界邮箱都是非阻塞的，而对于有界邮箱来讲，当消息数量达到邮箱容量，并且 mailbox-push-timeout-time 配置成非 0 时，会产生阻塞，表 4-1 是一部分内置邮箱的详细对比。

### 4.3.2 自定义优先级

在很多场景下，我们可能希望 Actor 可以根据设置好的优先级来处理消息，这些优先级通常和业务逻辑有关。为了能够自定义优先级，我们需要继承 PriorityMailbox 类并重写优先级规则，然后把该邮箱类型配置到配置文件，或者让 Actor 通过 withMailbox 的方式关联该邮箱。

下面来看看自定义消息优先级的具体实现过程。

首先，新建一个 class，继承 UnboundedPriorityMailbox 或者 UnboundedStablePriority-Mailbox：

表 4-1　内置邮箱及相关说明

| 邮箱类型 | 实现方式及说明 | 是否阻塞 | 是否有界 | 配置名称 |
| --- | --- | --- | --- | --- |
| UnboundedMailbox | 基于 java.util.concurrent.ConcurrentLinkedQueue 实现，先进先出的顺序 | 否 | 否 | unbounded 或者 akka.dispatch.Unb-ounded Mailbox |
| SingleConsumerOnly UnboundedMailbox | 基于一个多生产者单消费者的队列实现，不能使用在 Balancing-Dispatcher 调度器中 | 否 | 否 | akka.dispatch.Single ConsumerOnlyUnbounded Mailbox |
| NonBlockingBounded Mailbox | 基于一个多生产者单消费者的队列实现，会把超过 mailbox-capacity 的消息发送到 /deadLetters | 否 | 是 | akka.dispatch. NonBlockingBounded Mailbox |
| UnboundedControlAw areMailbox | 基于 java.util.concurrent.Concu-rrentLinkedQueue 实现，akka. dispatch. ControlMessage 类型的消息会被优先处理 | 否 | 否 | akka.dispatch. UnboundedControl AwareMailbox |
| UnboundedPriority Mailbox | 基于 java.util.concurrent.Priority-BlockingQueue 实现，可以通过重写该类型邮箱的排序方法自定义消息优先级，同一优先级的消息顺序并不固定 | 否 | 否 | akka.dispatch. UnboundedPriorityMailbox |
| UnboundedStable PriorityMailbox | 基于 java.util.concurrent.Priority-BlockingQueue 和 akka.util.Priority-QueueStabilizer 实现，对于同一优先级的消息会按照先进先出（FIFO）的顺序进行处理 | 否 | 否 | akka.dispatch.Unbounded StablePriorityMailbox |
| BoundedMailbox | 基于 java.util.concurrent.Linked-BlockingQueue 实现 | 当 mailbox-push-timeout-time 设置为非 0 时，可能会阻塞，反之则不会 | 是 | bounded 或者 akka.dispatch.BoundedMailbox |
| BoundedPriority Mailbox | 基于 java.util.PriorityQueue 和 akka.util. BoundedBlocking-Queue 实现，是 UnboundedPriority-Mailbox 的有界版本 | 当 mailbox-push-timeout-time 设置为非 0 时，可能会阻塞，反之则不会 | 是 | akka.dispatch.Bounded PriorityMailbox |
| BoundedStablePriority Mailbox | 基于 java.util.PriorityQueue、akka.util.Priority-QueueStabilizer 及 akka.util.Bounded-Blocking-Queue 实现，是 UnboundedStablePriority-Mailbox 的有界版本 | 当 mailbox-push-timeout-time 设置为非 0 时，可能会阻塞，反之则不会 | 是 | akka.dispatch.Bounded StablePriorityMailbox |
| BoundedControl AwareMailbox | 和 UnboundedControlAware-Mailbox 类似，akka.dispatch.Control-Message 类型的消息会被优先处理 | 当 mailbox-push-timeout-time 设置为非 0 时，可能会阻塞，反之则不会 |  | akka.dispatch.Bounded ControlAwareMailbox |

```
class MsgPriorityMailBox extends UnboundedStablePriorityMailbox {

    public MsgPriorityMailBox(ActorSystem.Settings settings, Config config) {
        /**
         * 返回值越小 优先级越高
         */
        super(new PriorityGenerator() {
            @Override
            public int gen(Object message) {
                if (message.equals("张三")) {
                    return 0;
                }else if(message.equals("李四")) {
                    return 1;
                }else if(message.equals("王五")) {
                    return 2;
                }else {
                    return 3;
                }
            }
        });}}
```

在该类中，需要定义一个带有 ActorSystem.Settings 和 Config 参数的构造方法，这个构造方法会在系统创建 MailBox 时反射调用。在构造方法中，需要给出一个 PriorityGenerator 对象，并重写 gen 方法，这个方法就是我们匹配消息优先级的地方，该方法返回值越小，则优先级越高。

然后，把该邮箱类型配置起来，如下：

```
msgprio-mailbox {
    mailbox-type = "com.javaakka.mailboxes.self.MsgPriorityMailBox"
}
```

让 mailbox-type 指向之前定义的邮箱类型。

测试时，让 Actor 通过 withMailbox 的方式关联该邮箱类型：

```
ActorSystem sys=ActorSystem.create("system");
ActorRef ref=
sys.actorOf(Props.create(MsgPriorityActor.class).withMailbox("msgprio-mailbox"),
    "priorityActor");
        Object[] messages= {"王五","李四","张三","小二"};
        for(Object msg:messages) {
            ref.tell(msg,ActorRef.noSender());
        }
```

MsgPriorityActor 是一个普通的 Actor，只是用来接收并打印消息，具体代码比较简单，在此略过。

执行测试代码，会得到如下结果：

```
张三
李四
王五
小二
```

结果说明：当我们给 MsgPriorityActor 发送消息后，它会通过自定义的 MsgPriority-MailBox 对消息进行优先级的排序。

### 4.3.3 控制指令优先

4.3.2 节中介绍的优先级队列，可以对整个消息进行排序，基本上能满足大部分需求。而对于有些比较特殊的关键消息，我们可能希望它每次都能拥有更高的优先级，这些消息通常来讲都是一些控制指令。Akka 提供了一种 ControlAwareMailbox 邮箱类型，可以让实现了 ControlMessage 接口的消息拥有最高优先级。

使用 ControlAwareMailbox 的第一步是定义消息，并实现 ControlMessage 接口：

```
class ControlMsg implements ControlMessage{
    private final String status;
    public ControlMsg(String status) {
        this.status=status;
    }
    @Override
    public String toString() {
        return this.status;
    }
}
```

ControlMessage 接口并没有需要实现的 API，仅仅是一个标志而已。

为了使用 ControlAwareMailbox，需要先进行如下配置：

```
control-aware-mailbox {
    mailbox-type = "akka.dispatch.UnboundedControlAwareMailbox"
}
```

最后，通过 withMailBox 方法让 Actor 关联该邮箱即可，下面是具体测试代码：

```
ActorSystem system=ActorSystem.create("sys");
ActorRef ref=
system.actorOf(Props.create(ControlAwareActor.class).withMailbox("control-aware-
    mailbox"),"controlAware");

Object[] messages= {"Java","C#",new ControlMsg("ServerPage"),"PHP"};
for(Object msg:messages) {
    ref.tell(msg,ActorRef.noSender());
}
```

其中 ControlAwareActor 也是一个普通的 Actor，只是用来接收并打印消息，具体代码在此略过。

执行这段代码后，会得到如下结果：

```
ServerPage
Java
C#
PHP
```

结果说明：实现了 ControlMessage 接口后的消息，在 ControlAwareMailbox 类型的邮箱中拥有最高优先级。

## 4.4 Actor 使用邮箱的多种方式

在上面使用不同邮箱的示例中，我们都是通过配置 mailbox 信息，然后让 Actor 调用 withMailbox 来关联指定的 mailbox，实际上，让一个 Actor 使用邮箱有多种方式，比如直接配置 Actor 的邮箱，或者将邮箱配置到 dispatcher，然后让 Actor 关联该 dispatcher 等，下面来看看这几种方式的具体做法。

### 4.4.1 配置 Actor 邮箱

当配置好 mailbox 信息后，可以不使用 withMailbox 来关联 mailbox，仅仅做如下配置就可以了：

```
akka.actor.deployment {
/priorityActor {
    mailbox = msgprio-mailbox
}}
```

其中 /priorityActor 是 Actor 的名字，mailbox 指向已经定义好的邮箱类型。

### 4.4.2 配置 dispatcher 邮箱

dispatcher 会通过邮箱进行消息调度，所以我们可以把邮箱配置在 dispatcher 上，然后让 Actor 关联该 dispatcher，配置如下：

```
my-msgprio-dispatcher {
    type = Dispatcher
mailbox-type = "com.javaakka.mailboxes.self.MsgPriorityMailBox"
    #其他dispatcher配置在此省略
}
```

Actor 关联 dispatcher 可以通过 withDispatcher 来实现，比如：

```
sys.actorOf(Props.create(MsgPriorityActor.class).withDispatcher("my-msgprio-dispa
    tcher"),"priorityActor")
```

也可以直接配置某个 Actor 的 dispatcher，具体内容可参考 3.3 节的内容。

### 4.4.3 实现 RequiresMessageQueue 接口

为了让 Actor 自动拥有某个特定类型的邮箱，可以让该 Actor 实现 RequiresMessage-Queue 接口，并且设置接口泛型为该邮箱队列的语义接口。从 Akka 的默认配置文件 reference.conf 中，我们可以得到如表 4-2 所示的对应关系。

表 4-2 邮箱队列语义接口及邮箱的对应关系

| 语义接口 | 邮箱类型 |
| --- | --- |
| akka.dispatch.UnboundedMessageQueueSemantics | akka.dispatch.UnboundedMailbox |
| akka.dispatch.BoundedMessageQueueSemantics | akka.dispatch.BoundedMailbox |
| akka.dispatch.DequeBasedMessageQueueSemantics | akka.dispatch.UnboundedDequeBasedMailbox |
| akka.dispatch.UnboundedDequeBasedMessageQueueSemantics | akka.dispatch.UnboundedDequeBasedMailbox |
| akka.dispatch.BoundedDequeBasedMessageQueueSemantics | akka.dispatch.BoundedDequeBasedMailbox |
| akka.dispatch.MultipleConsumerSemantics | akka.dispatch.UnboundedMailbox |
| akka.dispatch.ControlAwareMessageQueueSemantics | akka.dispatch.UnboundedControlAwareMailbox |
| akka.dispatch.UnboundedControlAwareMessageQueueSemantics | akka.dispatch.UnboundedControlAwareMailbox |
| akka.dispatch.BoundedControlAwareMessageQueueSemantics | akka.dispatch.BoundedControlAwareMailbox |
| akka.event.LoggerMessageQueueSemantics | akka.event.LoggerMailboxType |

以之前演示过的 UnboundedControlAwareMailbox 为例，假如通过 RequiresMessage-Queue 接口的方式，则应该编写如下代码：

```
public class CaActor extends UntypedActor implements
RequiresMessageQueue<UnboundedControlAwareMessageQueueSemantics> {
@Override
    public void onReceive(Object msg) throws Exception {
        System.out.println(msg);
    }
}
```

此时给该 Actor 发送的 ControlMessage 消息将会被优先处理。

除了使用内置的这些语义接口外，我们还可以自定义邮箱类型并配置语义接口 – 邮箱类型的对应关系。

## 4.5 自定义邮箱类型

在前面 4.3.2 节，我们了解到可以通过扩展某种特定类型的邮箱来实现一些个性化的需求，比如通过 PriorityMailbox 来实现自定义优先级，通过 ControlAwareMailbox 实现控制指令优先等，但是这些 DIY 并不彻底，比如：你不能通过这些接口直接修改队列的实现，也没办法对邮箱的消息做更精细化的控制。为了实现这些功能，我们可以直接自定义邮箱，

并且为该邮箱选择合适的队列实现。

自定义邮箱类型的过程是：首先定义一个邮箱队列（实现 MessageQueue 接口），然后定义邮箱类型（实现 MailboxType 接口），并指定邮箱队列，最后通过上面所讲的各种方式来关联到 Actor，下面我们通过代码来演示整个过程。

首先，定义一个类，取名 BusinessMsgQueue，让其实现 MessageQueue 接口：

```
class BusinessMsgQueue implements MessageQueue{
    private Queue<Envelope> queue=new ConcurrentLinkedQueue<Envelope>();
    @Override
    public void cleanUp(ActorRef owner, MessageQueue deadLetters) {
        for(Envelope el:queue) {
            deadLetters.enqueue(owner,el);
        }
    }
    @Override
    public Envelope dequeue() {
        return queue.poll();
    }
    @Override
    public void enqueue(ActorRef receiver, Envelope el) {
        queue.offer(el);
    }
    @Override
    public boolean hasMessages() {
    return !queue.isEmpty();
    }
    @Override
    public int numberOfMessages() {
        return queue.size();
    }
}
```

在 BusinessMsgQueue 中首先定义了 ConcurrentLinkedQueue 作为消息队列，然后实现了 MessageQueue 接口提供的 API，其中 enqueue 和 dequeue 分别表示投递消息（入队）和取出消息（出队）；假如在投递消息的过程中 Actor 不可用了，那么没有投递成功的消息会通过 cleanUp 进入死信队列。

定义邮箱需要实现 MailboxType 接口，并且在 create 方法中指定刚才定义好的队列：

```
class BusinessMailBoxType implements
MailboxType,ProducesMessageQueue<BusinessMsgQueue>{

    public BusinessMailBoxType(ActorSystem.Settings settings, Config config) {
    }

    @Override
    public MessageQueue create(Option<ActorRef> arg0, Option<ActorSystem> arg1) {
        return new BusinessMsgQueue();
    }
}
```

这里要注意的是，定义邮箱类型时必须给出一个公开的，带有 ActorSystem、Settings 这两个类型参数的构造方法，否则会创建失败。

邮箱类型创建好后，将其配置起来，然后就可以被 Actor 使用了：

```
business-mailbox {
    mailbox-type = "com.akkaaction.demo9.BusinessMailBoxType"
}
```

或者，也可以让该队列实现一个自定义的语义接口（任意接口即可，不需要有方法），并且把它映射到一个邮箱，假如某个 Actor 实现了带有该语义泛型的 RequiresMessage-Queue 接口，就会自动拥有该类型的邮箱，其中关键配置和代码如下：

```
#把语义接口映射到已经定义好的邮箱
akka.actor.mailbox.requirements {
"com.javaakka.mailboxes.self.MQSemantics" = business-mailbox
}

//让Actor实现RequiresMessageQueue接口，泛型为语义接口
class BusinessActor extends UntypedActor implements
RequiresMessageQueue<MQSemantics>
```

## 4.6 本章小结

每个 Actor 都有一个邮箱，用来存储其他地方发送过来的消息，依赖于底层队列的实现，这些消息将拥有不同的存放或取出顺序。Akka 内置了很多邮箱类型，基本上能满足大部分需求，这些邮箱类型大致分为两类：有界和无界，前者有容量限制，而后者无容量限制。假如这些邮箱类型仍不能满足需求，可以通过 MailboxType 和 MessageQueue 自定义邮箱。在项目中，我们需要根据实际情况来选用合适的邮箱类型，并进行合理的配置。

第 5 章 Chapter 3

# 消息路由

消息可以通过多种方式送达目的地，比如 tell、ask、forward 等，这些方式是最常规也是最简单的，但是对于复杂的消息投递逻辑，比如轮询投递、随机投递、广播组等，就需要开发者自己去做一层封装，好在 Akka 已经提供了丰富的路由组件，可以很好地满足这类需求。在实际项目中，我们通常会使用路由器来做负载均衡和任务分派。

在 Akka 中，使用路由的方式比较简单，基本上延续了普通 Actor 的常规操作，而且也提供了代码和配置两种路由创建方式，拥有极佳的灵活性和易用性。本章将全面介绍路由的基本用法和常规配置。

## 5.1 Router 和 Routee

在真正使用路由功能之前，我们需要了解几个常规的概念或词汇：Router 和 Routee。Router 表示路由器，消息会进入路由器转发出去，相当于消息的中转站，从程序实现上讲，它既可以是一个 Router 对象，也可以是一个 Actor 引用。Routee 表示路由目标，从路由器转发出来的消息会进入对应的路由目标，在使用前，需要通过配置或程序指定路由逻辑，如图 5-1 所示。

这里的路由逻辑主要是指以什么方式对消息进行投递，比如轮询、随机还是广播等。

## 5.2 路由器及路由逻辑

Akka 提供了两个原生语义的 API 来处理路由功能，它们分别是：akka.routing.Router 和



akka.routing.Routee。在使用它们时，需要先定义路由目标的 Actor，然后将它们包装成 Routee 对象，然后通过 Router 将消息根据路由逻辑转发给它们。下面我将用代码演示这个过程。

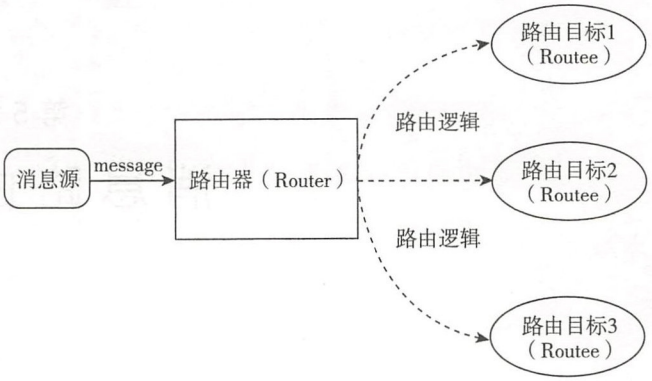



图 5-1　Router-Routee 运行流程

首先定义一个 Actor，作为 Routee：

```
class RouteeActor extends UntypedActor{
    @Override
    public void onReceive(Object msg) throws Exception {
        System.out.println(getSelf()+"-->"+msg);
    }
}
```

为了能更好地观察测试效果，这里把 getSelf() 打印出来。

然后定义一个 Actor，作为路由器的载体，用于转发消息：

```
class RouterTaskActor extends UntypedActor{
    private Router router;
    @Override
    public void preStart() throws Exception {
    List<Routee> listRoutee=new ArrayList<Routee>();
    for(int i=0;i<2;i++) {
    ActorRef
ref=getContext().actorOf(Props.create(RouteeActor.class),"routeeActor"+i);
    listRoutee.add(new ActorRefRoutee(ref));
    }
    router=new Router(new RoundRobinRoutingLogic(),listRoutee);
    }

    @Override
    public void onReceive(Object msg) throws Exception {
        router.route(msg,getSender());
    }
}
```



RouterTaskActor 启动后，会调用 preStart 方法（可参考生命周期），在该方法里，创建了两个 Routee。由于 Routee 类型不能直接使用，所以这里通过它的子类之一 ActorRefRoutee 来对 ActorRef 进行包装。在创建 Router 时，需要指定路由逻辑，即 RoutingLogic，RoundRobinRoutingLogic 表示通过轮询的方式给 Routee 发送消息。在实际应用中，我们很可能需要 watch 每个 Routee-Actor 的生命周期，一旦收到该 Actor 的终止信息（Terminated），应该将其移除出 Routee 列表。

最后，编写测试代码：

```
ActorSystem system=ActorSystem.create("sys");
ActorRef
routerActor=system.actorOf(Props.create(RouterTaskActor.class),
"routerTaskActor");
routerActor.tell("helloA",ActorRef.noSender());
routerActor.tell("helloB",ActorRef.noSender());
routerActor.tell("helloC",ActorRef.noSender());
```

通过 RouterTaskActor 发送消息后，会经过 Router 的 route 方法把消息轮流的转发给 RouteeActor。

该代码运行后结果如下：

```
Actor[akka://syst/user/routerTaskActor/routeeActor0#1744042123]-->helloA
Actor[akka://syst/user/routerTaskActor/routeeActor1#-927631075]-->helloB
Actor[akka://syst/user/routerTaskActor/routeeActor0#1744042123]-->helloC
```

当然，这里要注意，有可能打印出来的消息顺序并不完全是 A/B/C，这主要是因为多个 Actor 在接收同一个发送者的消息时是并行而不是串行的，但是每个 Actor 能保证接收到的消息是准确的，不会出现乱序接收的情况。

那么，RoundRobinRoutingLogic 到底是怎样来实现轮询投递的呢？通过它的源码我们可以发现，它继承自 RoutingLogic，并重写了 select 方法。在 select 方法中，它通过原子类 AtomicLong 来做递增，然后对 routees 的长度取余数，以此得到当前 routee 的下标，而在调用 Router 的 route 方法时，会通过如下 send 方法给目标发送消息。

```
send(logic.select(msg, routees), message, sender)
```

send 方法调用 RoutingLogic 的 select 方法，选中 Routee 后依赖 ActorRef 的 tell 来发送消息，具体过程可以参考图 5-2 所示。

除了 RoundRobinRoutingLogic 外，Akka 还提供多个其他类型的 RoutingLogic，比如 RandomRoutingLogic、BroadcastRoutingLogic、ScatterGatherFirstCompletedRoutingLogic 等，很显然，它们都有不同的 select 算法实现，部分常用 RoutingLogic 的具体使用方法我将在后面为大家介绍。



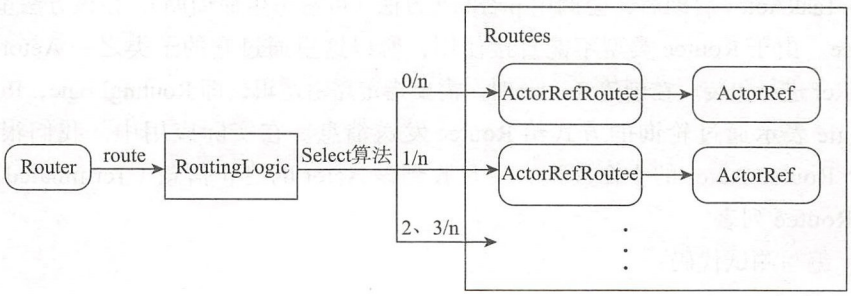



图 5-2 RoutingLogic——路由逻辑

## 5.3 路由 Actor

路由器可以是一个自包含的 Actor，它通常管理着自己的所有 Routee，一般来讲，我们会把这类路由配置在 *.conf 文件中，然后通过编码的方式加载并创建路由器。

创建一个路由 Actor 有两种模式：pool 和 group。

- pool 的方式表示路由器 Actor 会创建子 Actor 作为其 Routee 并对其监督和监控，当某个 Routee 终止时将其移除出去；
- group 的方式表示可以将 Routee 的生产方式放在外部（不必自包含），然后路由器 Actor 通过路径（path）对这些目标进行消息发送。

### 5.3.1 Pool 方式

使用 pool 方式定义的路由 Actor 会自动将 Routee 创建为自己的子级，这种层级关系在最开始就自动存在，不必通过 getContext.actorOf 的方式来指定。下面看看怎样实现一个 pool 的路由功能。

对于路由功能来讲，我们最好有一个消息中转的过程，即不会直接通过路由器来发送消息，而是先经过一层中间转发的过程，这样有利于构建更加清晰的管理结构，所以这里我们首先定义一个 Actor，作为消息中转处理器：

```
class MasterRouterActor extends UntypedActor {

    ActorRef router = null;

    @Override
    public void preStart() throws Exception {
        router = getContext().actorOf(
                new RoundRobinPool(3).props(Props.create(TaskActor.class)),
                "taskActor");
        System.out.println("router:"+router);
```

```
    }

    @Override
    public void onReceive(Object msg) throws Exception {
        router.tell(msg, getSender());
    }
}
```

在 MasterRouterActor 中，通过 RoundRobinPool 定义了 pool 类型的路由器（以及 Routee 池的大小），并指定了 Routee-Actor，这些 Actor 会自动成为 router 的子级。为了验证这个说法，我们可以在 TaskActor 直接打印出父级的信息：

```
class TaskActor extends UntypedActor {
    @Override
    public void onReceive(Object msg) throws Exception {
        System.out.println(getSelf() + "->" + msg+"-->"+getContext().parent());
    }
}
```

当给 MasterRouterActor 发送 helloA/helloB/helloC 消息后，打印结果如下：

```
router:Actor[akka://sys/user/masterRouterActor/taskActor#2049610389]
Actor[akka://sys/user/masterRouterActor/taskActor/$b#144271909]->helloB--
    >Actor[akka://sys/user/masterRouterActor/taskActor#2049610389]
Actor[akka://sys/user/masterRouterActor/taskActor/$c#-2061052157]->helloC--
    >Actor[akka://sys/user/masterRouterActor/taskActor#2049610389]
Actor[akka://sys/user/masterRouterActor/taskActor/$a#1073580911]->helloA--
    >Actor[akka://sys/user/masterRouterActor/taskActor#2049610389]
```

通过该结果可以看出，Routee 的父级就是自己的 Router。

在示例中，我们是通过硬编码的方式来创建 pool 路由的，实际上，Akka 还提供配置的方式来实现它，比如：

```
akka.actor.deployment {
    /masterRouterActor/taskActor {
     router = round-robin-pool
     nr-of-instances = 3
    }
}
```

其中 router 表示路由类型，nr-of-instances 表示 Routee 池的大小。配置完成后，需要将创建代码改为：

```
router = getContext().actorOf(
    FromConfig.getInstance().props(Props.create(TaskActor.class)),"taskActor");
```

即直接加载配置文件来创建路由。

到这里，大家可能会有个疑问，假如此时 Routee 回复一个消息会怎样呢？到底该谁接收？



实际上，路由和 forward 一样，在整个消息转发的过程中并不会改变原始 sender，所以消息会被回复给最初的 sender。并且，在回复消息时可以让父 Actor（路由 Actor）成为自己的 sender，这样在某种程度上可以隐藏自己的相关信息。

对于 pool 方式来讲，另外一个需要注意的是：由于父监督原则，路由 Actor 承担着 Routee 的监督工作，当没有显式指定监督策略时，路由 Actor 默认会把失败上溯到上级。当路由 Actor 重启时，会重新创建 Routee（子级 Actor），并且在池中维护相同个数（nr-of-instances）的 actor；当所有的 Routee 被终止时，路由 Actor 也会停止（watch 它的生命周期，就可以收到它的 Terminated 消息）。

假如需要自定义监督策略，可以通过如下方式创建并进行绑定：

```
SupervisorStrategy strategy = new OneForOneStrategy(3,
        Duration. create( "1 minute"), new Function<Throwable, Directive>() {
            @Override
            public Directive apply(Throwable t ) {
                //代码略
            }
        })
ActorRef router=getContext().actorOf(
    new RoundRobinPool(3).withSupervisorStrategy(strategy ).props(Props.create
        (TaskActor. class)), "taskActor" );
```

## 5.3.2 Group 方式

当我们需要单独（外部）创建 Routee 时，可以采用 group 方式。在使用 group 路由前，需要先定义 Routee-Actor，然后将它们以 path 的形式配置起来，路由 Actor 会通过 path 列表来进行路由。下面是路由 Actor 的核心代码：

```
//定义Routee
getContext().actorOf(Props.create(WorkTask.class), "wt1");
getContext().actorOf(Props.create(WorkTask.class), "wt2");
getContext().actorOf(Props.create(WorkTask.class), "wt3");
router=getContext().actorOf(FromConfig.getInstance().props(),"router");
```

在确定了 Aactor 的层级关系或者 path 之后，需要将它们配置到 conf 文件中：

```
akka.actor.deployment {
/masterActor/router {
        router = round-robin-group
        routees.paths = ["/user/masterActor/wt1", "/user/masterActor/wt2","/
            user/masterActor/wt3"]
    }
}
```

其中 /masterActor/router 表示路由器的 path，routees.paths 用来配置 Routee 的 path，并且本地和远程 Actor 都是支持的。假如希望这些 path 能更加动态地产生（或者依赖其他业

务逻辑产生），可以使用编码的方式来实现，比如：

```
List<String> routeePaths = Arrays.asList("/user/masterActor/wt1", "/user/
    masterActor/wt2","/user/masterActor/wt3");
router=getContext().actorOf(new RoundRobinGroup(routeePaths).props(),"router");
```

还有一点要注意的就是：group 方式并不限制 Routee 一定得在同一个层级下，比如当你想增加一个其他外部的 Actor 作为 Routee 时，仅仅需要将它的 path 配置在 routees.paths 中即可。

## 5.4 常见路由类型

为了满足不同场景的需求，Akka 提供了多种路由类型供我们使用，表 5-1 是部分路由类型的汇总说明。

表 5-1 部分路由类型的功能描述

| 名 称 | 功 能 说 明 | 配 置 名 称 |
| --- | --- | --- |
| RoundRobinRouting | 轮询的给每个 Routee 发送消息，它支持 RoundRobin-Pool 和 RoundRobinGroup 两种形式 | round-robin-pool 或 round-robin-group |
| RandomRouting | 每次会随机的给某一个 Routee 发送消息，它支持 RandomPool 和 RandomGroup 两种形式 | random-pool 或 random-group |
| BalancingRouting | 它会将繁忙的 Routee 中的任务重新分配给比较空闲的 Routee，所有的 Routee 会共享同一个邮箱。它不支持 group 形式 | balancing-pool |
| SmallestMailbox | 它会试着给消息较少的非挂起的 Routee 发送消息。它不支持 group 形式 | smallest-mailbox-pool |
| Broadcast | 它会以广播的形式发送给所有 Routee，它支持 Broadcast-Pool 和 BroadcastGroup 两种形式 | broadcast-pool 或 broadcast-group |
| ScatterGatherFirst Completed | 它会将消息发送给所有 Routee，并期待一个最快的回复（此时其他回复会被丢弃），假如在设定的时间内不能得到一个回复，则会产生一个 akka.pattern.AskTimeoutException。它支持 ScatterGatherFirstCompletedPool 和 ScatterGather-FirstCompletedGroup 两种形式 | scatter-gather-pool 或 scatter-gather-group |
| TailChopping | 它会首先发送消息给一个随机选取的 Routee，间隔短时间后发给第二个随机选取的 Routee，以此类推。它也期望一个最快的回复（其他回复被丢弃）。这种做法可以降低延迟的风险，毕竟“没把鸡蛋放在同一个篮子里”。它支持 TailChoppingPool 和 TailChoppingGroup 两种形式 | tail-chopping-pool 或 tail-chopping-group |
| ConsistentHashing | 它使用一致性 Hash 算法来选择 Routee。它依赖三种方式来定义一致性 Hash 的 key：通过 withHashMapper 指定路由的 ConsistentHashMapper；让消息实现 akka.routing.ConsistentHashingRouter.ConsistentHashable 接口；把消息包装在 akka. routing.ConsistentHashingRouter.Consistent-HashableEnvelope 中。它支持 ConsistentHashing-Pool 和 ConsistentHashingGroup 两种形式 | consistent-hashing-pool 或 consistent-hashing-group |



下面，我们来演示一下 Broadcast、ScatterGatherFirstCompleted 和 TailChopping 的具体用法。

### 5.4.1 广播 -Broadcast

广播路由器会给所有指定的 Routee 发送消息，即每个 Routee 会收到同样的消息。为了方便演示，我后面统一采用 Group 的形式来创建路由。

首先，可以定义任意两个 Routee-Actor，用来接收路由过来的消息，核心代码如下：

```
class BroadWorker1 extends  UntypedActor{
    @Override
    public void onReceive(Object msg) throws Exception {
        System.out.println(getSelf()+"-->"+msg);
    }
}
class BroadWorker2 extends  UntypedActor{
        //代码略
}
//多个路由Routee-Actor ...
```

BroadWorker1 和 BroadWorker2 作为 Routee-Actor 存在，可以有任意多个。然后创建 Routee 和 Router：

```
class MasterBroadActor extends UntypedActor{
    private ActorRef router;
    @Override
    public void preStart() throws Exception {
        getContext().actorOf(Props.create(BroadWorker1.class),"bw1");
        getContext().actorOf(Props.create(BroadWorker2.class),"bw2");
router=getContext().actorOf(FromConfig.getInstance().props(),
"broadRouter");
    }

    @Override
    public void onReceive(Object msg) throws Exception {
        router.tell(msg, getSender());
    }
}
```

这里要注意，把 Group 类型的 Routee 创建为某个 Actor 的子级不是必须的，你可以将它创建为顶级 Actor（/user 下），只不过为了监管和层次划分的需要，可能很多时候会这样创建而已。当所有 path 都能确定后，需要做如下配置：

```
akka.actor.deployment {
    /masterBroadActor/broadRouter {
     router = broadcast-group
     routees.paths = ["/user/masterBroadActor/bw1","/user/masterBroadActor/bw2"]
    }
}
```



其中 /masterBroadActor/broadRouter 表示匹配的路由 Actor 的路径，router 指定了 broadcast-group 类型，routees.paths 指定了 Routee 的 path。最后，可以用如下代码进行测试：

```
ActorSystem system=ActorSystem.create("sys");
ActorRef master=system.actorOf(Props.create(MasterBroadActor.class),"masterBroadActor");
master.tell("helloA", ActorRef.noSender());
master.tell("helloB", ActorRef.noSender());
```

运行代码，会得到如下结果：

```
Actor[akka://sys/user/masterBroadActor/bw1#-1930033020]-->helloA
Actor[akka://sys/user/masterBroadActor/bw2#73080028]-->helloA
Actor[akka://sys/user/masterBroadActor/bw1#-1930033020]-->helloB
Actor[akka://sys/user/masterBroadActor/bw2#73080028]-->helloB
```

很显然，该路由器给每个 Routee 发送了同样的消息。

### 5.4.2 最快响应 -ScatterGatherFirstCompleted

ScatterGatherFirstCompleted 将消息发送给所有 Routee，并且会等待一个最快的响应，一旦收到一个响应回复，其他回复将会被丢弃。和之前介绍的所有路由一样，首先得定义 Routee，下面是部分演示代码：

```
getContext().actorOf(Props.create(FCActor1.class),"fw1");
getContext().actorOf(Props.create(FCActor2.class),"fw2");
router=getContext().actorOf(FromConfig.getInstance().props(),"firstCompRouter");
```

FCActor1、FCActor2 分别是两个普通的 Actor，用来接收消息并通过“getSender().tell("A",getSelf());”的方式回复消息（分别回复 A、B），由于代码比较简单，这里不再贴出。

ScatterGatherFirstCompleted 路由的本意是将一个业务处理（比如查询或计算）分发到不同的节点，让最快的节点处理并返回消息，以此降低延迟，提高吞吐率，但是假如在这个过程中发生错误，导致节点并不能正常返回消息，那么业务发起者是否需要一直等待呢？很显然不是。

ScatterGatherFirstCompleted 提供了 within 参数，用于设置最长等待时间，在这个时间段内，如果发起者不能收到任何一个正常回应，则会收到一个 time out 的错误提示，下面是参考配置：

```
akka.actor.deployment {
    /masterFirstActor/firstCompRouter {
     router = scatter-gather-group
     routees.paths = ["/user/masterFirstActor/fw1",
"/user/masterFirstActor/fw2"]
     within = 3 seconds
    }
}
```



为了能接收到 Routee 返回的消息，可以采用 ask 的方式发起业务，然后通过 Future 回调的方式处理消息：

```
Timeout timeout= new Timeout(Duration.create(10, "seconds"));
Future<Object> fu=Patterns.ask(master, "helloA", timeout);
fu.onComplete(new OnComplete<Object>() {

        @Override
        public void onComplete(Throwable err, Object result)
                throws Throwable {
            System.out.println("err: "+err);
            System.out.println("result: "+result);
        }
    }, system.dispatcher());
```

Future.onComplete 会异步执行，当它收到回复后，会通过 onComplete 来处理异常和消息。大家可以自行在 FCActor1、FCActor2 的 onReceive 方法中设置 sleep 代码，模拟耗时的操作，运行结果后会发现，你总是能接收到最快响应的消息。如果设置的 sleep 时间超过了 within 设置的时间，则会在 onComplete 中收到一个 akka.pattern.AskTimeoutException，此时 result 为 null。

### 5.4.3 随机 – 最快响应 -TailChopping

TailChopping 是一个比较有趣的路由类型，它会首先将消息发送给一个随机选取的 Routee，在延迟一小段时间后将消息发送给第二个随机选取的 Routee，以此类推。和 ScatterGatherFirstCompleted 类似，它也期望得到第一个响应并回给原始 sender，其他回复则被丢弃。

实际上，我们可以很方便地将上面 ScatterGatherFirstCompleted 的示例改造成 TailChopping，要做的仅仅是在配置文件中配置 router 类型和发给下一个随机 Routee 的消息的间隔时间，如：

```
/masterFirstActor/firstCompRouter {
    router =  tail-chopping-group
    routees.paths = ["/user/masterFirstActor/fw1","/user/masterFirstActor/fw2"]
    within = 5 seconds
# 发给下一个routee的时间间隔
    tail-chopping-router.interval = 1 seconds
}
```

### 5.4.4 创建可修改容量的池

在之前的示例中，无论是通过配置还是硬编码，我们创建的都是固定容量的 pool，即该容量在程序运行时无法动态改变。有时候，我们可能更希望 pool 能够根据实际业务量动

态的增加或降低容量，此时可以考虑使用 Resizer 和 OptimalSizeExploringResizer 功能。

### 1. Resizer

Resizer 可以根据路由的繁忙程度来决定增加还是降低 pool 的容量，当它发现处理消息的压力大于某个阈值时会增加容量，当它发现处理消息的压力小于某个阈值时则会降低容量。使用 Resizer 功能的方式比较简单，仅仅配置如下内容即可：

```
/masterResizeActor/resizeRouter {
        router = round-robin-pool
        resizer {
            lower-bound = 2
            upper-bound = 4
            messages-per-resize = 10
    }
}
```

其中 lower-bound 和 upper-bound 分别表示最小和最大 routee 的数量，upper-bound>= lower-bound。messages-per-resize 表示在每次执行 resize（修改容量）间隔期间所处理的消息数目。

### 2. OptimalSizeExploringResizer

OptimalSizeExploringResizer 会持续的对 pool 进行优化，以便让 pool 保持在一个最佳容量。一般认为，它比 Resizer 更耗内存。下面是它的核心配置：

```
akka.actor.deployment {
    /masterResizeActor/resizeRouter {
     router = round-robin-pool
optimal-size-exploring-resizer {
            enabled = on
            action-interval = 3s
        downsize-after-underutilized-for = 10h
        downsize-ratio = 0.9
    }
}}
```

其中 action-interval 表示每次 resize 操作的间隔时间，downsize-after-underutilized-for 表示当所有 routee 都没有满额任务的持续时间超过这个时间长度时，就会将容量值降低到一个合理的值，该值也与 downsize-ratio 配置相关，比如说原本最高使用 20 个 routee，现在会降到 18 个。

## 5.5 特殊消息处理

正常情况下，发送给路由的消息都会按照指定的路由策略转发给 Routee，但是对于某些特殊的消息，它的处理方式可能有所不同，下面我们来看看这些“异类”。

### 5.5.1 Broadcast 消息

在 5.4.1 节中，笔者介绍过 Broadcast 路由的使用，该类型的路由会给每个 Routee 发送相同的消息。实际上，我们还可以通过直接给某路由发送 Broadcast 消息来实现类似的功能，比如：

```
router.tell(new Broadcast("broad cast"), ActorRef.noSender());
```

此时该路由会给所有 Routee 发送“broad cast”（注意：是被包装的消息而不是 Broadcast 对象）的消息，并且这种行为与原本的路由策略无关。

### 5.5.2 PoisonPill 消息

正常情况下，给一个 Actor 发送 PoisonPill 消息后，该 Actor 会被终止，这对路由也是适用的。但是发送给路由的 PoisonPill 消息会直接被路由自己消化，而不会被转发给 Routee。对于 pool 类型的路由，或者其他含有层级关系的路由策略（即 Routee 被创建为某个 Router 的子级），当父级收到 PoisonPill 消息时，会在自己终止前先终止子级，子级会在处理完“手头”上的消息后正式终止，而邮箱队列里面的消息将得不到处理。通常来讲，这并不是一个好的终止策略（当然，除非是由于某些特殊需求而不得不这样做），更好的做法是：直接让 Routee 收到终止的消息，让它能在停止之前继续处理其他消息。这时候，前面介绍的 Broadcast 就能派上用场了，比如：

```
router.tell(new Broadcast(PoisonPill.getInstance()),ActorRef.noSender());
```

在给 Router 发送包装了 PoisonPill 的 Broadcast 消息后，每个 Routee 都会收到 PoisonPill 消息，此时 PoisonPill 会作为一个普通的消息进入邮箱队列，然后在所有现有消息处理完之后再来处理它，这样可以确保已有的消息都能得到处理。

和 PoisonPill 类似的另外一个消息类型是 Kill，它能杀死一个 Actor，同时子 Actor 也会被杀掉，对于路由来讲也会存在上述问题，所以建议将 Kill 包装在 Broadcast 中：

```
router.tell(new Broadcast(Kill.getInstance()), getSender());
```

### 5.5.3 其他管理类消息

管理类消息包括新增 Routee，删除 Routee，得到现有 Routee 列表等，表 5-2 是部分汇总说明。

表 5-2 管理类消息及其说明

| 消息类型 | 说　明 |
| --- | --- |
| akka.routing.AddRoutee | 向路由发起新增 Routee 指令，它需要封装一个已经定义的 Routee（比如 ActorRefRoutee） |

（续）

| 消息类型 | 说明 |
| --- | --- |
| akka.routing.RemoveRoutee | 向路由发起删除 Routee 指令，它也需要封装一个已经定义的 Routee（比如 ActorRefRoutee） |
| akka.routing.GetRoutees | 向路由发起查询所有 Routee 的指令，路由会返回一个 Routees 对象，该对象包含了所有 Routee 的引用 |
| akka.routing.AdjustPoolSize | 向 pool 类型路由发起改变池大小的请求，它需要一个 int 型参数，正数表示新增 N 个 Routee，负数表示删除 N 个 Routee |

下面是部分演示代码：

```
Timeout timeout= new Timeout(Duration.create(10, "seconds"));
//新增一个Routee
ref.tell(new AddRoutee(new ActorRefRoutee(tmpRef)), ActorRef.noSender());

//删除一个Routee
ref.tell(new RemoveRoutee(new ActorRefRoutee(tmpRef)), ActorRef.noSender());

//得到Routee列表
Future<Object> fu=Patterns.ask(ref, GetRoutees.getInstance(), timeout);
    fu.onComplete(new OnComplete<Object>() {
    @Override
    public void onComplete(Throwable err, Object result)
            throws Throwable {
        Routees rs=(Routees)result;
        List<Routee> listRoutees=rs.getRoutees();
        for(Routee r:listRoutees) {
            System.out.println("routee:"+r);
        }
    }
}, system.dispatcher());
```

该代码有两点需要注意：

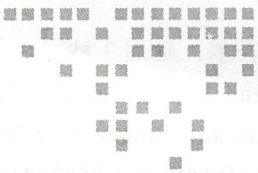
1）使用同一个 ActorRef 包装的两个 ActorRefRoutee，实际上描述的还是一个 Routee；

2）新增或删除 Routee 这类操作可能并不会马上生效，需要在发送 GetRoutees 消息后根据返回的 Routees 情况才能判断出来是否已经变更。

## 5.6 本章小结

给一个 Actor 发送消息是非常简单的，仅仅通过 tell 或 ask 的方式就可以实现，但是给一组 Actor 进行有规则地发送消息则需要考虑很多问题，比如路由逻辑、监督和管理等，

而 Akka 提供的路由组件已经帮我们实现了其中大部分功能。在 Akka 中，路由包含两种形式：pool 和 group，前者创建带有父子层级关系的路由器，后者则创建根据任意 path 寻址 Routee 的路由器。为了满足不同需求，Akka 提供了多种路由类型，比如给所有 Routee 发送消息的广播路由，期望得到最快响应的 ScatterGatherFirstCompleted 等，而且这些路由都有编码和配置两种创建方式，使用起来非常方便。



第 6 章 Chapter 6

# 实用工具

在前面章节中，笔者介绍了 Akka 的基础知识及用法，相信大家已经能够编写简单的 Akka 应用了。为了更好地使用 Akka，我们需要掌握更多的工具或配置。在本章中，笔者将为大家介绍下面这些工具：

- Scheduler，提供定时调度功能；
- Future，用于接收 Actor 并发操作的结果；
- EventBus，事件总线、事件流；
- Logging，日志处理，包括日志配置，slf4j 等；
- Extensions，Akka 扩展，比如怎样让它支持 Spring；
- Akka I/O，基于 TCP/UDP 的网络服务；
- Akka Streams，基于流的数据处理工具。

## 6.1 定时调度——Scheduler

Akka 提供了 Scheduler 对象来帮助我们实现定时调度的功能。Scheduler 主要有两类 API：scheduleOnce 和 scheduler，前者表示延迟一段时间后执行，后者表示延迟后定时执行，并且拥有取消定时执行的能力。Scheduler 对象不是凭空 new 出来的，而是需要通过 Actor.scheduler() 得到，它在整个 Actor 系统内都是单例的。实际上，在 Actor 系统启动后，会读取 akka.scheduler.implementation 这个配置项，默认获取到的 Scheduler 正是由此生成，当然，你也可以自定义该实现。下面先看看 scheduleOnce 的示例：

```
system.scheduler().scheduleOnce(Duration.create(3, "s"), refTarget,
```

```
"hello",system.dispatcher(), refSender);
```

这段代码表示：延迟 3 秒之后给 refTarget 发送一条消息（“hello”），该 Acor 返回的消息会被 refSender 收到，假如不需要收到回复，则可以将最后一个参数设置为 null（即无发送者）。Duration.create 方法第二个参数的可选值有：day，hour，minute，second，millisecond 等，分别表示不同时间尺度。

schedule 方法示例如下：

```
Cancellable cancelHandler=system.scheduler().schedule(Duration.create(1, "s"),
    Duration.create(3, "s"), refTarget, "hello",system.dispatcher(), refSender);
//其他处理过程略...
cancelHandler.cancel();
```

这段代码表示：延迟 1 秒之后给 refTarget 发送一条消息（“hello”），然后每隔 3 秒执行一次（发消息）。在不需要该定时任务时，可以调用 cancel 方法将其取消。

## 6.2 处理并发结果——Future

为了得到某个 Actor 的响应，通常会调用 receive 方法来接收对方返回的数据，但是这种操作必须发生在 Actor 之间，假如需要在外部也能够接收某个 Actor 的响应数据，可以使用 Future API。使用 Future API 的另外一个好处是，它允许我们通过多种方式来处理消息，比如同步等待结果、异步回调、组合计算、管道处理等。

### 6.2.1 Future 常规用法

使用 Future 的最简单方式就是通过 Patterns.ask 方法接收一个 Future 对象，然后使用同步或异步的方式去处理消息。同步的方式主要是通过 Await.wait 或 ready 方法阻塞等待返回值来实现，异步主要通过各种回调函数实现。需要注意的是：在实现消息传递时，我们必须考虑到超时带来的影响，很多时候，超时的设置和处理往往影响着整个系统的吞吐率和可用性，所以在使用 ask 和 future 之前，必须设定一个合适的超时时间。下面是相关示例代码：

```
public class TestFuturesActor {
    public static void main(String[] args) {
    ActorSystem system = ActorSystem. create( "sys");
    ActorRef ref = system .actorOf(Props.create(FutureActor. class ),
    "fuActor" );
    Timeout timeout = new Timeout(Duration.create(3, "seconds" ));
    Future<Object> future = Patterns. ask( ref, "hello future", timeout);
    try {
    String replymsg = (String) Await.result( future, timeout.duration());
    System. out .println(replymsg );
    } catch (Exception e ) {
```

```
        e.printStackTrace();
        }
        }}
class FutureActor extends UntypedActor {
        @Override
        public void onReceive(Object msg ) throws Exception {
        //Thread.sleep(4000);
        getSender().tell( "reply" , getSelf());
        }
}
```

在示例代码中，首先通过 Duration API 创建了用于限制超时的 Timeout 对象，然后通过 Patterns.ask 向 Actor 异步发送消息，并接收一个 scala.concurrent.Future 对象，该对象里面封装了 FutureActor 返回的消息。在接收到 Future 之后，一般会采用两种方式来处理消息：同步（阻塞）或异步（非阻塞）。代码中的 Await.result 即同步 API，也就是说，只有接收到数据后，才会继续往下执行，假如超时了（把 sleep 解开），则会抛出 java.util.concurrent.TimeoutException。

假如要异步处理消息，则需要显式地依赖 ExecutionContext 对象，并且调用回调函数。Future 提供的回调函数有：onComplete、onSuccess 和 onFailure，它们分别表示完成、成功、失败的处理，下面是示例代码：

```
future .onSuccess( new OnSuccess() {

        @Override
        public void onSuccess(Object msg) throws Throwable {
        System. out .println( "receive: "+ msg);
        }

        }, system .dispatcher());

        future .onFailure( new OnFailure() {

        @Override
        public void onFailure(Throwable ex ) throws Throwable {
        if (ex instanceof AskTimeoutException) {
            System. out .println( "超时异常");
        } else {
            System. out .println( "其他异常 "+ ex);
        }
    }
}, system .dispatcher());
```

当成功获取消息时，会进入 onSuccess 回调函数，并且将消息传给 OnSuccess 方法；当出现异常时（比如超时），会进入 onFailure 回调函数，并且将异常传给 onFailure 方法，除了这两个函数外，还可以使用 onComplete 函数，它相当于把 onSuccess 和 onFailure 的功能整合在一起，大家可以自行尝试。

### 6.2.2 函数式 Future

函数式编程由来已久，并不是一个新的概念，在很多语言中都有它的影子。精心设计过的函数式 API 往往具有结构简单、功能明确的特点，并且有着强劲的算法描述能力。Future 提供了非常好用的函数式 API，可以让我们更方便地对数据进行处理或计算。

#### 1. map 函数

假如需要对 Future 包含的数据进行处理，并能返回新的值，可以使用 map 函数。map 函数本身也会返回一个新的 Future，在后面可以继续对 Future 的其他常用操作：

```
Future<String> f2= f1.map( new Mapper<Object,String>() {
    @Override
    public String apply(Object parameter ) {
    return ((String)parameter ).toUpperCase();
    }
    }, system.dispatcher());
f2.onSuccess...
```

map 函数需要一个 Mapper 对象，该对象的两个泛型分别是输入与输出的引用类型，与 apply 方法的参数和返回值类型对应。通过 map 得到新值后，之前 Future 中的值是不受影响的。

map 函数不仅是用在单个 Future 里面，还可以用在被序列操作之后的 Future 列表之上，也就是说，当你通过某种方式得到一系列 Future 之后，你可以尝试对它们做一些有趣的计算，比如统计。在实际项目中，一个很常见的场景是：将计算分配给多个单元去并行处理，然后汇总所有的结果并做计算，最后返回给请求发起方。这里笔者通过一个简单的例子来说明该如何实现类似功能，这个例子需要实现：统计一系列单词的长度并求总和。

首先，定义一个 Actor 处理用于单个计算：

```
class CountActor extends UntypedActor {

    @Override
    public void onReceive(Object msg ) throws Exception {
    String message = (String) msg;
    getSender().tell( message.length(), getSelf());
    }
}
```

然后创建多个 Actor 实例，对每个 Actor 实例发起计算，返回的多个 Future 记录到 List：

```
String[] words = { "hello" , "akka" , "future" };
    ActorSystem system = ActorSystem. create( "sys");
    Timeout timeout = new Timeout(Duration.create(3, "seconds" ));
    List<Future<Integer>> list = new ArrayList<Future<Integer>>();
    for (int i = 0; i < words .length ; i ++) {
        ActorRef countActor = system .actorOf(
        Props. create(CountActor. class), "countActor" + i );
```

仅供非商业用途或交流学习使用

```
        Future<Integer> future = (Future) Patterns.ask( countActor,
        words[i], timeout) ;
        list. add( future);
}
```

但是此时的 List 还不能直接做 map 操作，需要先通过 sequence 让其变成一个包含结果列表的 Future 对象，此时 Future 列表中的数据会被传入 apply 方法中：

```
ExecutionContext context= system.dispatcher();
Future<Iterable<Integer>> fs = Futures.sequence( list, context);
Future<Integer> fv = fs.map( new Mapper<Iterable<Integer>, Integer>() {

@Override
public Integer apply(Iterable<Integer> plist ) {
    int total = 0;
    for (Integer v : plist ) {
    total = total + v;
    }
    return total ;
    }

    }, system.dispatcher());
    try {
    Integer total = (Integer) Await. result( fv, timeout .duration());
    System. out .println("结果是: " + total );
    } catch (Exception e ) {
    e.printStackTrace();
}
```

在 apply 中循环出每个值，然后做统计处理，并把新的值包装成新的 Future，后面就可以通过常规方式得到这个值了。

### 2. fold 函数

fold 函数提供了初始值的参数，当一个 Future 列表传入后，首先通过初始值开始计算，并且将计算结果作为下一次计算的开始，核心代码如下所示：

```
Future<Integer> fs = Futures. fold(10, list , new Function2<Integer, Integer,
Integer>() {
    @Override
    public Integer apply(Integer v1, Integer v2 ) throws Exception {
    System. out .println(v1 +"" +v2 );
    return v1 +v2 ;
    }
}, system.dispatcher());
```

fold 函数会在最后一个 Future 执行完之后开始执行。它有两个重要的参数，第一个参数表示初始值，它将在第一次执行处理时传给 apply；第二个参数 Function2 用来处理每个数据，并生成下一次的初始数据，它的三个泛型分别对应 apply 方法两个参数的引用类型及

仅供非商业用途或交流学习使用

返回值类型。

假如你并不需要一个初始值的参数，那么可以使用 reduce 函数，它的用法和 fold 类似，只要去掉第一个参数就可以了。

## 6.3 事件总线

一个日益复杂的业务总会带来架构上的挑战，其中一个很重要的原因是，这些业务中总会存在各种依赖。在购物网站中，一个典型的场景是：当用户成功购买某件商品后，商品库存会自动更新。而“库存的更新”这件事往往也是其他系统所关心的，比如商家系统希望能看到某件商品的剩余库存，用户系统希望收到某个关注的商品的库存情况，后台系统希望能统计商品的库存变化。按照常规做法，需要在库存更新后，给每一个已知的“关心者”发送通知消息，假如后续再有新的“关心者”，必须要再回头去添加通知代码。也就是说，按照这种做法，必须要先知道“关心者”的存在，这在某种程度上增加了系统之间的耦合度，而事件总线为解决这类问题提供了一个很好的思路。

事件总线并不复杂，它延续了观察者模式的思想，实现了基于发布—订阅的消息流处理。以上面所讲的场景为例，在一个购物网站中，“库存的更新”往往被很多其他业务“关心”（依赖），同时也有可能被“关心者”放弃（解除依赖），那么怎么才能实现这些依赖关系的解耦呢？实际上，最好的做法就是“发布—订阅”模式。我们可以让“关心者”订阅“库存的更新”事件，然后在库存真正更新时，发布这个事件，那么此时“关心者”就会收到该通知，当订阅者不再需要依赖这类事件时，可以自行取消订阅，整个过程中，发布者是不需要知道订阅者的存在的。

实际上，这个模式在互联网中随处可见，比如当你在网上看到一篇感兴趣的文章，你可能会订阅该博客，下次作者发布新文章时，你会收到更新提醒，此时博客作者是不需要知道你是谁的，同时，你也有取消订阅的权利。

### 6.3.1 实现事件总线

在 Akka 中，事件总线被抽象成 EventBus 类型，它拥有发布（publish）、订阅（subscribe）、取消订阅（unsubscribe）等功能。具体来讲，一个完整的消息总线的过程是：发布者将 Event 发布到 EventBus 上，事件订阅者（Subscriber）将会接收到相应的通知消息，通常来讲，订阅者会是一个 Actor，它通过原有的 onReceive 方法接收该消息。在这个过程中，还有一个概念非常重要，那就是 Classifier，它用来描述一个事件分类，不同事件类型将有不同的订阅者，EventBus 将会通过 Classifier 来选择订阅者并向其发送消息。常见的 Classifier 有 LookupClassification、SubchannelClassification 等，前者指定匹配的事件分类，由每个类别维护一个订阅者列表，后者可以匹配层级的事件分类。

仅供非商业用途或交流学习使用

下面我们来看看怎样实现一个 LookupClassification 分类的 EventBus。首先定义一个事件实体，拥有事件类型与事件消息：

```
class Event{
    private String type;
    private String message;
    public Event( String type, String message) {
    this .type = type ;
    this .message = message ;
    }
    public String getType() {
    return type ;
    }
    public String getMessage() {
    return message ;
    }
}
```

然后定义一个 class，继承 LookupEventBus，实现 classify、publish 等关键逻辑：

```
class EventBusDemo extends LookupEventBus<Event, ActorRef, String>{

    @Override
    public String classify(Event event) {
    return event .getType();
    }

    @Override
    public int compareSubscribers(ActorRef ref1 , ActorRef ref2 ) {
    return ref1.compareTo( ref2);
    }

    @Override
    public void publish(Event event, ActorRef ref) {
    ref.tell( event.getMessage(),ActorRef. noSender());
    }

    /**
    * 期望的classify数,一般设置为2的n次幂
    */
    @Override
    public int mapSize() {
    return 8;
    }

}
```

LookupEventBus 需要三个泛型，分别指定事件类型、订阅者类型、Classifier 类型。当该 EventBus 发布 Event 时，将会先通过 classify 方法来选择目标订阅者，然后采用恰当的

仅供非商业用途或交流学习使用

方式通知他们，当然，这一步也是对开发者开放的（就像 publish 方法做的那样）。

Actor 在订阅 Event 时，需要调用 EventBus 的 subscribe 方法，该方法会在内部维护一个 Map 结构，key 为 classify，value 为订阅者列表，在订阅的过程中，它会通过 compareSubscribers 方法来做排序比较。下面是订阅与发布的部分测试代码：

```
//新建EventBus
    EventBusDemo bus= new EventBusDemo();

    //订阅info与warn事件
    bus.subscribe( eventSubActor , "info" );
    bus.subscribe( eventSubActor , "warn" );

    //发布info事件
    bus.publish( new Event("info" ,"Hello EventBus" ));

    //取消订阅
    bus.unsubscribe( eventSubActor ,"warn" );

    //发布warn事件
    bus.publish( new Event("warn" ,"Oh No" ));
```

其中 eventSubActor 是一个普通的 Actor 实例，它订阅了 info 和 warn 两个事件，当 EventBus 通过 publish 发布 info 事件时，它会收到“Hello EventBus”的消息；由于在发布 warn 事件之前，它取消了该事件的订阅，所以不会收到“Oh No”的消息。

### 6.3.2 事件流处理

事件流是 ActorSystem 上下文环境中的事件总线，使用它可以很方便地将“发布—订阅”模式用于任意事件类型——它既可以是自定义的类型，也可以是死信（Dead Letter）或系统日志。它的订阅者通常都是 Actor，当 Actor 被终止后，它会自动将其从订阅者列表中移除。

在 Akka 中，事件流对象使用 EventStream 类型来描述，它继承自 LoggingBus，实现了 SubchannelClassification 分类器，所以可以满足层级事件的订阅需求。EventStream 通常由 ActorSystem 得到，使用起来非常简单，就像下面这样：

```
system.eventStream().subscribe(actorRef, DeadLetter.class);
```

该行代码给 actorRef 订阅了 DeadLetter 事件，当给某个已经终止的 Actor 发送消息后，该消息会进入死信列表，然后 actorRef 会收到 DeadLetter 消息。通过 DeadLetter 消息，我们可以得到死信内容、发送者、接收者：

```
public void onReceive(Object msg) throws Exception {
    if(msg instanceof DeadLetter) {
    DeadLetter deadLetter=(DeadLetter)msg;
    System.out.println("deadLetter:"+deadLetter.message()+"
"+deadLetter.sender()+""+deadLetter.recipient());
```

仅供非商业用途或交流学习使用

```
    }else {
    System.out.println("otherMessage"+msg);
    }
}
```

常规的死信消息都能通过上面这种方式收到提醒，但是DeadLetterSuppression消息除外，实现了该接口的死信消息将不会被默认的监听日志记录到，所以无法通过订阅DeadLetter来收到提醒。假如你需要这类死信的提醒，应该订阅SuppressedDeadLetter或者AllDeadLetters，前者只作用于DeadLetterSuppression消息，而后者作用于所有消息。

刚才笔者说过，EventBus实现了SubchannelClassification分类器，因此它可以包含层级类型的匹配，这使得我们能够订阅一个父类下的所有事件。比如，我们有一个父级事件SuperMessage，然后有另外两个子事件MessageA和MessageB，它们的关系如下所示：

```
interface SuperMessage{
}
class MessageA implements SuperMessage{
    //内部代码略
}
class MessageB implements SuperMessage{
    //内部代码略
}
```

当某个Actor订阅MessageA事件后，是收不到MessageB的消息的，但是假如订阅的是SuperMessage，则无论是MessageA还是MessageB都能收到：

```
system.eventStream().subscribe(ref,SuperMessage.class);
system.eventStream().publish(new MessageB("t"));
system.eventStream().publish(new MessageA("d"));
```

## 6.4 日志处理

在Akka中，日志会以异步的方式进行，以降低性能损耗。在默认情况下，日志会被打印到标准输出，当然，你可以自定义这种行为。在2.3.1节中，我们曾通过如下代码创建一个日志对象：

```
LoggingAdapter log = Logging.getLogger(getContext().system(),this);
```

事实上，第一个参数还可以传递一个LoggingBus对象，比如getContext().system().eventStream()。LoggingAdapter对象包含debug、info、warning、error等方法，它们对应不同的日志等级，并且都能通过{}占位符来指定参数：

```
log.info("receive result {},continue... ",message);
new String[] {"one","two","three"}
log.warning("receive numeber:{},{},{}", message);
```

仅供非商业用途或交流学习使用

### 6.4.1 基础配置

在 Akka 系统运行的各个场景，基本上都提供了日志记录，非常便于我们及时了解系统状况，表 6-1 是部分场景的配置说明。

表 6-1 日志配置说明

| 配 置 项 | 解 释 |
| --- | --- |
| akka.loglevel | 日志等级，可选值有 OFF、ERROR、WARNING、INFO、DEBUG，默认为 INFO |
| akka.loggers | 配置日志监听实现，默认是 akka.event.Logging$DefaultLogger，它会把日志输出到控制台。当然，也可以自定义或使用第三方监听器，比如 slf4j |
| akka.log-config-on-start | 设置为 on 表示在启动阶段打印出当前使用的完整配置 |
| akka.log-dead-letters | 记录 DeadLetter 的信息数量，默认为 10。可选值有 on、off、数值。on 表示记录所有 DeadLetter，off 表示不记录 DeadLetter |
| akka.log-dead-letters-during-shutdown | 在 ActorSystem 停止阶段记录 DeadLetter 日志信息，默认为 on，off 表示不记录任何日志 |
| akka.actor.debug.autoreceive | 设置为 on 表示记录所有自动接收的消息（DEBUG 等级下），比如 PoisonPill 消息 |
| akka.actor.debug.lifecycle | 设置为 on 表示记录所有生命周期的变化（比如启动、重启、停止等）。 |
| akka.actor.debug.event-stream | 设置为 on 表示记录 ActorSystem.eventStream 的订阅情况。 |

### 6.4.2 使用 slf4j

在生产环境中，并不推荐使用默认的日志监听器，很多时候都会自定义日志处理，或者使用 slf4j 这样的第三方组件。自定义监听器非常简单，仅仅需要针对不同级别的日志信息做区分处理就行了，比如：

```
class LogListener extends UntypedActor {

    @Override
    public void onReceive(Object msg) throws Exception {

    if (msg instanceof InitializeLogger) {
    System.out.println("init: "+msg);
    getSender().tell(Logging.loggerInitialized(), getSelf());
    } else if (msg instanceof Error) {
    System.out.println("error: "+msg);
    } else if (msg instanceof Warning) {
    System.out.println("warn: "+msg);
    } else if (msg instanceof Info) {
    System.out.println("info: "+msg);
    } else if (msg instanceof Debug) {
    System.out.println("debug: "+msg);
    }else {
```

仅供非商业用途或交流学习使用

```
        System.out.println("unhandled: "+msg);
        }
        }
}
```

然后将该 class 配置在 *.conf 文件中即可：

```
akka.loggers = ["com.javaakka.eventbus.self.LogListener"]
```

此时通过 LoggingAdapter 记录的日志，将会被包装成 Debug、Info、Warning、Error 消息传给 LogListener，在 LogListener 中可以根据匹配到的日志消息做不同的处理。这里要注意，默认情况下 akka.loglevel=INFO，直接运行这段代码是收不到 debug 信息的，大家需要设置 akka.loglevel=DEBUG 才行。

目前市面上已经有很多日志的“轮子”了，我们不必再去造一个，在这里，笔者推荐使用 slf4j 日志组件。slf4j 对 Java 程序员来说并不陌生，它提供了一个日志的整体框架，而不负责具体实现，假如要使用该组件，最好将其和 logback 配合使用，下面是相关依赖：

```
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-slf4j_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.1.3</version>
</dependency>
```

然后配置 loggers：akka.loggers = ["akka.event.slf4j.Slf4jLogger"]，此时再通过 LoggingAdapter 记录的日志，就会委托给 slf4j 和 logback 来处理。当然，按照 logback 的常规用法，我们应该还要配置 logback.xml 文件，并通过一些辅助变量得到更有用的信息：

```
<appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
        <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} %X{sourceThread}
-> %X{akkaSource} - %msg%n
</pattern>
    </encoder>
</appender>

<root level="error">
    <appender-ref ref="stdout" />
</root>
```

sourceThread 变量在这里表示线程名称，akkaSource 变量表示 Actor 的地址，它们的输出看起来是这样的：

仅供非商业用途或交流学习使用

```
2016-09-18 17:18:06 ERROR c.j.eventbus.self.LogDemoActor sys-akka.actor.default-
    dispatcher-2 -> akka://sys/user/logDemoActor - hello
```

在实际项目中，为了能更加精确地记录系统运行状况，可能会希望添加一些自定义的业务变量进入日志，此时可以使用 DiagnosticLoggingAdapter 来代替普通的 LoggingAdapter，它通过设置一个 Map 结构来注入变量值：

```
DiagnosticLoggingAdapter log = Logging.getLogger(this);
Map<String, Object> logparam = new HashMap<String, Object>();
logparam.put("user", "afei");
logparam.put("role", "manager");
log.setMDC(logparam);
//...
//debug info warning error 代码省略...
//...
log.clearMDC();
```

现在，在日志 pattern 里面就可以用 %X{user} 和 %X{role} 来引用 map 里面设置的变量值了。最后，在这个示例中，笔者仍然是使用了控制台输出的方式来处理日志，假如需要将日志输出到文件里，可以采用 ch.qos.logback.core.rolling.RollingFileAppender，具体用法大家可以参考 LogBack 官方文档。

## 6.5 Akka 扩展

在复杂多变的业务中，抽取一些公共的组件是很有必要的，一般情况下，我们会尽量将这些组件做成“插拔式”的，即在不影响当前系统或框架的情况下，满足某些定制化的需求。在这方面，Akka 提供了一种非常好的扩展组件的方式，即 Akka Extensions（扩展）。实际上，Akka 内部有很多组件就是基于扩展实现的，比如 TypedActor、序列化、集群指标等，接下来，我们先以 TypedActor 为例，来看看 Akka 扩展大致是怎样使用的。

### 6.5.1 TypedActor 初探

在 Akka 中，Actor 主要分为两类：UntypedActor 和 TypedActor。UntypedActor 更能体现经典 Actor 模型的优势，所以更加常用。而 TypedActor 的使用方式比较接近 OOP，即通过接口—实现类和函数调用的方式来驱动任务的执行，这对部分 Java 开发者来讲可能更容易接受，在实际项目中，尽量不要因为简单而滥用。

#### 1. 使用 TypedActor

TypedActor 实际上是作为 Akka 的 Extension 而存在的，所以在使用时，必须先通过 ActorSystem 得到一个 TypedActorExtension 对象，然后通过该对象创建需要被包装的接口的代理。也就是说，使用 TypedActor 的一个前提是，必须要先有业务接口和实现类，然后才能将其包装成一个可异步执行的 API。当然，在底层执行的时候，依然会依赖 ActorRef

来实现，只不过对调用层屏蔽了这个细节。

接下来，笔者用一个示例来为大家介绍到底该如何创建并使用 TypedActor，这个示例实现的功能非常简单：提供一个对用户信息操作的 API 接口，要求该接口拥有新增、查询等功能，并且可以异步调用。

首先，我们必须定义好对外提供的接口和实现：

```
public interface UserService{

public void saveUser(String id,String user);

public Future<String> findUserForFuture(String id);

public Option<String> findUserForOpt(String id);

public String findUser(String id);

}

class UserServiceImpl implements UserService{

    private static Map<String,String> map=new ConcurrentHashMap<String,
        String>();

    @Override
    public void saveUser(String id, String user) {
        map.put(id, user);
    }

    @Override
    public Future<String> findUserForFuture(String id) {
        return Futures.successful(map.get(id));
    }

    @Override
    public Option<String> findUserForOpt(String id) {
        return Option.some(map.get(id));
    }

    @Override
    public String findUser(String id) {
        return map.get(id);
    }
}
```

UserService 业务接口拥有 4 个方法，1 个专门用于添加用户信息的 saveUser 方法和 3 个用于查询用户信息的 find* 方法，在实现类 UserServiceImpl 中使用 Map 模拟 DB 操作并实现了所有接口方法。可能有部分人觉得奇怪：为什么某些接口方法需要使用 Future 或 Option 接口作为返回类型？实际上，这是因为不同的返回类型会直接影响方法的运行策略，

这个策略的规则是：

- 当方法无返回值（即 void）的时候，底层会采用 ActorRef.tell 的方式来调用，执行方式为异步；
- 当方法返回 scala.concurrent.Future 时，会以 Patterns.ask 的方式来调用，然后将结果值包装到 Future 里并返回，它的执行方式也是异步；
- 当方法返回 akka.japi.Option 时，会以 Patterns.ask 的方式来调用，但是程序会一直阻塞，直到有返回值。假如返回值为 null，它会被包装成 Option.None 类型；
- 当方法返回其他类型时，就和普通方法一样，程序会一直阻塞，直到有返回值。

此时，接口和实现类不能按照常规方式去使用，否则就失去了 TypedActor 的意义。使用该接口的正确方式是，首先让 TypedActorExtension 生成它的代理，然后调用其业务方法。这些方法将被 JDK 代理拦截，最终委托底层 Actor 去执行，下面是部分测试代码：

```
//得到TypedActor扩展对象
TypedActorExtension typeActorExtension=TypedActor.get(system);

//得到接口代理实现
UserService userService=typeActorExtension.typedActorOf(new TypedProps<UserServi
    ceImpl>(UserService.class,UserServiceImpl.class));
    System.out.println("userService: " + userService);
    //异步执行?
    userService.saveUser("1","afei");
    //异步执行
    Future<String> fu = userService.findUserForFuture("1");
    fu.onSuccess(new OnSuccess() {
    @Override
    public void onSuccess(Object result) throws Throwable {
    System.out.println("The future user is:" + result);
    }
    }, system.dispatcher());

    //阻塞直到有返回值
    Option<String> opt=userService.findUserForOpt("1");
    System.out.println("The Opt user is:"+opt.getClass());

    //阻塞直到有返回值
    String user=userService.findUser("1");
    System.out.println("The user is: "+user);
```

在示例中，我们直接向 TypedProps 中传入了实现类的类型，此时它会调用默认的构造方法来创建对象，假如希望调用含有参数的其他构造方法，则需要传入 Creator 接口实现，然后自己构造实现类对象：

```
new TypedProps<UserServiceImpl>(UserService.class, new
    Creator<UserServiceImpl>() {
        @Override
```

仅供非商业用途或交流学习使用



```
        public UserServiceImpl create() throws Exception {
            return new UserServiceImpl("Akka");
        }
});
```

另外，通过typedActorOf得到的UserService对象，实际上已经是一个被普通Actor包装之后的代理实现（通过打印userService即可看出），这个“影子Actor”可以通过typeActorExtension.getActorRefFor方法得到。

### 2. TypedActor API

除了上面所讲的基本用法外，TypedActor还提供了一些工具性质的API供我们使用，表6-2列出了部分API及说明：

表6-2 TypedActor提供的部分工具API说明

| API | 说 明 |
| --- | --- |
| TypedActorExtension.isTypedActor | 判断某个对象是否被TypedActor代理过 |
| TypedActorExtension.getActorRefFor | 得到底层运行的Actor |
| TypedActorExtension.stop | 异步的终止指定的TypedActor |
| TypedActorExtension.poisonPill | 在完成当前任务后异步地终止指定的TypedActor |
| TypedActor.PreStart、TypedActor.PostStop、TypedActor.PreRestart、TypedActor.PostRestart | 实现这些接口的preStart、postStop等方法即可获取TypedActor的生命周期状况 |
| TypedActor.context() | 得到当前ActorContext对象，该方法只能在TypedActor实现类方法里使用 |
| TypedActor.self() | 得到外部代理对象，该方法只能在TypedActor实现类方法里使用 |

## 6.5.2 自定义扩展

Akka的扩展需要依赖Extensions API，它包含两个概念：Extension和ExtensionId。Extension表示一个可用的扩展组件，它对每一个ActorSystem都是唯一的，即每次通过同一个ActorSystem得到的Extension对象都是同一个。ExtensionId是用来创建及查找Extension的组件，将其配置在*.conf中可以让ActorSystem在创建阶段加载Extension。

为了方便大家理解自定义扩展的步骤，这里我们先实现一个简单示例，然后再通过扩展将Spring和Akka集成。接下来我们将要演示的示例场景是：某个ActorSystem需要和外界做RPC调用，而远程服务地址和端口是可配置的。

首先，我们需要在*.conf文件中定义配置项：

```
akkademo{
    server=serverdemo.io
    port=1234
}
```

仅供非商业用途或交流学习使用



该配置包含RPC的server地址以及端口，它们会在加载时传入Extension对象，Extension的实现如下：

```
public class RPCExtension implements Extension {
    private String server;
private int port;
    public RPCExtension(String server, int port) {
    this.server = server;
    this.port = port;
    }
    //模拟RPC调用
    public void rpcCall(String cmd) {
    System.out.println("call "+cmd+"-->"+server+":"+port);
    }
}
```

它实现了Extension接口，其他地方和普通Java Class无异。为了使该类对象能被ActorSystem查找并绑定，需要再定义一个ExtensionId类型：

```
public class RPCExtProvider extends AbstractExtensionId<RPCExtension>{

    private static RPCExtProvider provider=new RPCExtProvider();

    @Override
    public RPCExtension createExtension(ExtendedActorSystem system) {
    Config config=system.settings().config();
    String server=config.getString("akkademo.server");
    int port=config.getInt("akkademo.port");
    return new RPCExtension(server,port);
    }

    public static RPCExtProvider getInstance() {
    return provider;
    }
}
```

该类继承了AbstractExtensionId并实现了createExtension方法，该方法主要用于Extension组件的创建，它的ExtendedActorSystem参数即当前环境下的ActorSystem对象引用，所以可以直接通过它来获取Config对象，进而读取配置信息。在使用它查找Extension时，需要调用AbstractExtensionId提供的get方法：

```
RPCExtension rpcExt=RPCExtProvider.getInstance().get(system);
rpcExt.rpcCall("hello");
```

当调用get方法时，RPCExtProvider.createExtension会被自动调用一次，假如将该类配置在akka.extensions里，该方法会在ActorSystem创建阶段自动调用一次。

仅供非商业用途或交流学习使用



### 6.5.3 集成 Spring

Spring 是 Java 世界中最为著名的一款开源框架，它以 IoC 和 AOP 为基础，提供了从 MVC 到 DB 层的一站式功能，并且由于 Spring Boot 和 Spring Cloud 的兴起，使得它在微服务中也得到广泛的应用。在本节中，笔者将介绍 Akka 集成 Spring 的常规方法。

#### 1. 依赖注入

在 2.3.3 节，笔者介绍过可以通过 Props/Creator 的方式自定义 Actor 工厂，除此之外，Akka 还提供了专门的依赖注入（DI）方案，可以让 Akka-Actor 很方便地和其他 Bean 容器框架（比如 Spring）进行集成。DI 功能通过 IndirectActorProducer 接口实现，示例代码如下：

```
class DI implements IndirectActorProducer{
    @Override
    public Class<? extends Actor> actorClass() {
        return ActorDemo.class;
    }
    @Override
    public Actor produce() {
        return new ActorDemo();
    }
}
```

在 DI 类中，我们重写了 actorClass 和 produce 两个方法，它们分别返回 Class 类型和 Actor 实例对象。在创建 Actor 时，需要将 DI 传入 Props：

```
ActorRef ar =sys.actorOf(Props. create(DI. class), "actorDemo" );
```

很多时候，我们会在 produce 方法中调用某个“bean 工厂”来得到一个 ActorBean，很显然，通过这种方式来集成 Spring 将是极好的。同时要注意的是：假如集成了某个 DI 框架，那么 Actor 不能被配置成单例。

#### 2. 集成 Spring

集成 Spring，主要是为了将 ActorSystem 和 Actor 的创建、Actor 系统与其他系统组件的依赖都纳入 Spring 的管理中。在做集成之前，我们有必要先理清一下整个过程。对于 ActorSystem 来说，它并不是使用 new 的方式来创建的，而是拥有自己的工厂方法，即 create，假如要将其设置为一个 Spring Bean，需要在配置时指定 factory-method，但是这种方式并不是太简洁，官方推荐采用 @Configuration 的方式来生产 ActorSystem；对于 Actor 来说，它是通过 actorOf 方法来创建的，在该方法中，可以传入前面所讲的依赖注入组件 IndirectActorProducer，然后让 ApplicationContext.getBean 方法返回 Actor；一般来说，在集成了 Spring 后，所有 ActorSystem 范围内的依赖都会交给 Spring 来管理，并且每个 ActorSystem 都会持有一个 ApplicationContext，所以可以将集成功能做成 Akka 的扩展。综上所述，我们可以采用图 6-1 给出的方案来集成 Spring。

仅供非商业用途或交流学习使用



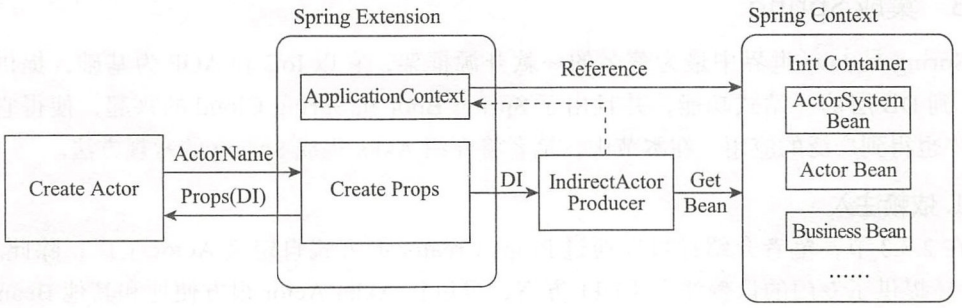



图 6-1 Akka 集成 Spring 的解决方案

图 6-1 中的 Spring Extension 表示 Akka-Spring 扩展，它持有一个 ApplicationContext 引用，以便在 IndirectActorProducer 创建 Actor 时直接从 Spring 容器中 getBean，当 Spring Extension 拿到创建 Actor 的 Props 对象时会返回给客户程序，此时客户程序就可以通过 ActorSystem 或 ActorContext 的 actorOf 方法生成 Actor 了。

接下来，我们用编码实现图 6-1 的整个过程。在这之前，请先确保项目中已经加入以下依赖：

```
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>3.2.4</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>3.1.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>3.1.1.RELEASE</version>
</dependency>
```

首先，我们需要定义 IndirectActorProducer，作为生产 Actor 的工厂，它需要依赖两个参数：Spring Bean 工厂（ApplicationContext）和 Actor 的名字：

```
class SpringDI implements IndirectActorProducer{

    private ApplicationContext applicationContext;

    private String beanName;

    public SpringDI(ApplicationContext applicationContext,String beanName) {
    this.applicationContext=applicationContext;
    this.beanName=beanName;
```

仅供非商业用途或交流学习使用



```
    }

    @Override
    public Class<? extends Actor> actorClass() {
    return (Class<? extends Actor>) applicationContext.getType(beanName);
    }
    @Override
    public Actor produce() {
    return (Actor) applicationContext.getBean(beanName);
    }
}
```

此时使用 SpringDI 构造的 Props 来创建 Actor 的话，就能通过 ApplicationContext 拿到 Spring 容器内的 Actor Bean 了，构造 Props 的过程可以放到自定义的扩展里：

```
public class SpringExt implements Extension {

    private ApplicationContext applicationContext;

    public void initApplicationContext(ApplicationContext applicationContext) {
    this.applicationContext=applicationContext;
    }

    public Props createProps(String actorBeanName) {
    return Props.create(SpringDI.class,this.applicationContext,actorBeanName);
    }
}

public class SpringExtProvider extends AbstractExtensionId<SpringExt>{

        private static SpringExtProvider provider=new SpringExtProvider();
        @Override
        public SpringExt createExtension(ExtendedActorSystem system) {
            return new SpringExt();
        }

        public static SpringExtProvider getInstance() {
            return provider;
        }

}
```

其中 SpringExt 是扩展组件，它持有的 ApplicationContext 对象将在 Spring 加载时被传进来，SpringExtProvider 是一个 ExtensionId，通过它可以在 ActorSystem 范围内创建并查找 SpringExt。createProps 方法主要用来构造 Props，它依赖于前面创建的 IndirectActorProducer，一旦在外部得到该 Props，就可以创建基于 Spring 容器内的 Actor Bean 了。那么，我们该怎样初始化 ActorSystem 并扫描到被纳入容器的 Actor 呢？ Spring 提供了一种简单方法来实现 Bean 的扫描，即 @Configuration 和 AnnotationConfigApplicationContext，通过 AnnotationCon-

仅供非商业用途或交流学习使用



figApplicationContext 扫描到的被注解的 Class，都能被纳入 Bean 容器。为了创建 AcorSystem 并初始化，可以按照下面这种方式定义一个配置类：

```
@Configuration
public class AppConfig {

    @Autowired
    private ApplicationContext applicationContext;

    @Bean
    public ActorSystem createActorSystem() {
    ActorSystem system=ActorSystem.create("sys");
    SpringExtProvider.getInstance().get(system).
initApplicationContext(applicationContext);
    return system;
    }
}
```

这个类主要做了两件事：创建 ActorSystem 对象，并纳入 Spring 容器管理；初始化 SpringExt 中的 ApplicationContext。对于 Actor 来讲，仅仅将其作为普通的 Bean 就好，这里仍然通过注解来实现 Bean 的定义：

```
@Component("actorDemo")
@Scope("prototype")
public class ActorDemo extends UntypedActor {

    @Autowired
    private EmpService empService;

    @Override
    public void onReceive(Object msg) throws Exception {
    System.out.println("receive: "+msg);
    empService.saveEmp((String)msg);
    }
}
```

这里要注意，在定义 Actor Bean 时，务必要将其设置为非单例的状态，否则程序会出问题。在这个 Actor 里面，EmpService 就是一个普通的业务 Bean，具体代码不再给出。

在使用扩展来创建 ActorSystem 和 Actor 前，需要先确保所有的 Bean 都被扫描到，下面是部分测试代码：

```
    AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
    //扫描package
    ctx.scan("com.javaakka.bean");
    ctx.refresh();
    ActorSystem system=ctx.getBean(ActorSystem.class);
    ActorRef
```

仅供非商业用途或交流学习使用



```
ref=system.actorOf(SpringExtProvider.getInstance().
get(system).createProps("actorDemo"),"actorDemo");
    ref.tell("hello",ActorRef.noSender());
```

## 6.6 Akka I/O

Akka I/O 模块提供了一套完全基于事件驱动、异步非阻塞的传输层 API，为实现高性能网络应用提供了坚实的基础。Akka I/O 支持 TCP 和 UDP 两种传输协议，其中 TCP 是面向连接的传输协议，也就是说在通信前必须与对方建立连接，比较适合对可靠性要求高的应用，而 UDP 是面向非连接的传输协议，可靠性不如 TCP。Akka I/O 的 API 都是基于 Actor 的，这意味着所有操作（比如连接或数据传输）都需要依赖于消息传递而不是方法调用。

### 6.6.1 TCP 服务

在 Akka I/O 中，实现 TCP 的第一步是获取其管理器 Actor，即 TcpManager。TcpManager 可以接收两类指令：akka.io.Tcp.Bind 和 akka.io.Tcp.Connect，当向 TcpManager 发送 Bind 指令时，可以绑定和监听一个指定的地址及端口号，即创建一个 Tcp 服务；当向 TcpManager 发送 Connect 指令时，可以打开一个 Tcp 连接。TcpManager 会随着 TcpExt（一个扩展）而启动，我们可以通过如下代码得到它：

```
ActorRef tcpManager = Tcp.get(system).manager();
```

#### 1. TCP 服务端

如前面所说，当你想要构造一个 TCP 服务时，你首先要做的是向一个 TcpManager Actor 发送 Bind 指令，待绑定成功后会收到 Bound 消息，表示已经准备好接受客户端的连接。当客户端向 TcpManager 发送 Connect 指令时，客户端和服务端都会接收到 Connected 消息，该消息中包含了两端的地址和端口信息，接下来就可以互相通信了。下面代码演示了如何创建一个 TCP 服务端，笔者将在后面讲解它的整个运行过程：

```
class TcpServerDemo extends UntypedActor {
    @Override
public void preStart() throws Exception {
    super.preStart();
ActorRef tcpManager = Tcp.get(getContext().system()).manager();
tcpManager.tell(TcpMessage.bind(getSelf(), new
InetSocketAddress("127.0.0.1", 1234), 100), getSelf());
    }
    @Override
    public void onReceive(Object msg) throws Exception {
        if(msg instanceof Bound){
            Bound bound=(Bound)msg;
```

仅供非商业用途或交流学习使用



```
            System.out.println("bound:"+bound);
            /*..其他处理略..*/
        }else if(msg instanceof Connected){
            Connected conn=(Connected)msg;
            System.out.println("conn:"+conn);
            ActorRef handler =
            getContext().actorOf(Props.create(ServerHandler.class));
            getSender().tell(TcpMessage.register(handler),getSelf());
        }}}
class ServerHandler extends UntypedActor {
    @Override
    public void onReceive(Object msg) throws Exception {
    if (msg instanceof Received) {
        Received re = (Received) msg;
        ByteString b = re.data();
        String content = b.utf8String();
        System.out.println("server:"+content);
        ActorRef conn=getSender();
        conn.tell(TcpMessage.write(ByteString.fromString("Thanks")),getSelf());
    } else if (msg instanceof ConnectionClosed) {
        System.out.println("Connection is closed "+msg);
        getContext().stop(getSelf());
    }else {
        System.out.println("Other Tcp Server: " + msg);
    }}}
```

当 TcpServerDemo 启动时，首先通过 TcpExt 创建并获取 TcpManager Actor，该 Actor 的名字为 IO-TCP。然后使用 TcpMessage 创建 Bind 指令，并将其发送给 TcpManager，在后面你将会看到，大部分的指令都是由 TcpMessage 创建的。TcpMessage.bind([handler, endpoint,backlog]) 方法有三个参数，它们的含义如下：

- handler，指定了哪个 Actor 将会接收所有接入的连接请求（Connected）；
- endpoint，指定了连接的地址，包括 IP 和端口号；
- backlog，指定了未处理连接队列的最大容量，超过这个数后会拒绝连接。

### 2. 绑定及注册

当 Bind 指令被发送到 TcpManager 后，它会创建一个 TcpListener Actor，并在该 Actor 内部监控 Bind 的 handler（即 TcpMessage.bind 的第一个参数），然后监听指定的端口。假如 bind 成功，该消息的发送者（即 sender）会收到来自 TcpListener Actor 回复的 Bound 消息，到这一步即表明 Tcp 服务已经准备就绪了。

假如此时客户端发起了连接请求，服务端会收到 Connected 消息，根据惯例，我们应该得到一个代表“连接”的对象才能继续与客户端通信，那么这个“连接”对象从哪里来呢，是从 Connected 里来吗？

实际上，Connected 消息的发送者（sender）就是我们需要的连接对象，它是一个 Actor（也称为 Connection Actor），我们可以直接通过 getSender() 来引用它，并通过它与对方进

仅供非商业用途或交流学习使用



行通信。这里要注意，在使用连接对象之前，我们还得注册一个用于处理服务端消息的Actor，否则连接对象是不可用的。连接对象负责监控注册的Actor，当这个Actor终止时，连接对象会关闭连接并释放相关资源。假如在一定时间内没有完成注册，连接对象将会终止，该时间由akka.io.tcp.register-timeout设定，默认为5秒。

### 3. 接收消息

当客户端向服务端发送数据时，服务端会接收到Received消息，该消息中携带了客户端发过来的具体数据。这个数据是ByteString类型的，它是一个高效的不可变字节容器，比较适合用于Actor通信。为了得到可读的字符串数据，需要调用其utf8String方法进行转换。

### 4. 关闭连接

当客户端给连接对象发送TcpMessage.close()消息时会关闭当前连接，关闭成功之后客户端会收到Closed消息，服务端会收到PeerClosed消息。在服务端发送该消息时也会遵循同样的处理流程。假如其中某一端强制关闭连接，对方会收到ErrorClosed消息。Closed、PeerClosed、ErrorClosed消息都继承自ConnectionClosed，所以可以统一使用ConnectionClosed进行关闭处理。需要注意的是，当连接被关闭时，我们应该将处理连接的Actor关掉以释放资源。

### 5. TCP客户端

客户端在连接服务端时也需要先得到TcpManager Actor，然后向其发送Connect消息，当连接成功后，消息发送者（sender）会收到Connected消息，下面是部分示例代码：

```
ActorRef tcp = Tcp.get(getContext().system()).manager();
InetSocketAddress address = new InetSocketAddress("127.0.0.1", 1234);
tcp.tell(TcpMessage.connect(address), getSelf());
```

和服务端类似，客户端Connected消息的发送者就是连接对象（即Connection Actor），当客户端收到Connected消息后，可以通过getSender()引用这个Actor。在真正使用这个连接对象之前，我们还需要注册一个用于处理客户端消息的Actor，然后才能通过连接对象向服务端发送消息：

```
if (msg instanceof Connected) {
// Connection Actor
ActorRef conn = getSender();
ActorRef clientHandler =
getContext().actorOf(Props.create(ClientHandler.class), "clientHandler");
    conn.tell(TcpMessage.register(clientHandler), getSelf());
     conn.tell(TcpMessage.write(ByteString.fromString("HelloAkka")),getSelf());
}
```

ClientHandler是一个普通的Actor，它的处理方式和前面服务端的ServerHandler类似，即处理来自服务端的消息（比如Received），这里不再给出代码。很多时候，我们可能会直

接注册当前 Actor（即 getSelf()）作为处理 Actor，这样做的好处是方便管理，比如当连接关闭时可以调用 getContext().stop(getSelf()) 来一次性关闭客户端 Actor。而对服务端更多的时候会注册一个新的 Actor 来处理每个连接，因为不太可能在某个连接关闭后就直接关掉整个服务。

#### 6. 读写控制

除了实现正常的“请求—响应”之外，Akka I/O 还提供了更多手段来精细化控制读写过程。比如当你在向服务端发送数据时，可以采用 ACK-based 模式，即发送数据时携带一个 ACK 数据（可以理解为确认数据），当写入成功后会收到这个 ACK 数据，这样我们就可以在之前的数据写入之后再发送下一段数据。或者在读数据的时候发现处理消息时压力过大，可以向连接对象发送 TcpMessage.suspendReading() 消息通知对方，此时它会暂停接收新数据，当处理压力比较平缓时，则可以向连接对象发送 TcpMessage.resumeReading() 消息，以恢复数据接收。下面代码演示了如何定义一个 ACK 类型，以及如何携带 ACK 数据：

```
public class MyAck implements Event {
    private int ack;
    public MyAck(int ack) {
        this.ack = ack;
    }
}
//携带ACK数据
MyAck ack = new MyAck(23);
conn.tell(TcpMessage.write(ByteString.fromString("Hello
Java"),ack),getSelf());
```

实际上，我们之前在没有携带 ACK 数据时，TcpMessage.write 方法会默认携带 NoAck 数据。

### 6.6.2 UDP 服务

UDP 是面向非连接的协议，即客户端在和服务端进行通信时是不必与对方先建立连接的，在 Akka I/O 中，UDP 服务和 TCP 服务的用法是很类似的，比如它们都需要在服务端向管理器 Actor 发送 Bind 信息进行端口绑定和监听，而在通信时也需要通过向连接对象（Connection Actor）发送特定消息来进行数据传输。下面我们简单看一下 UDP 的实现过程。

首先，在服务端，我们需要实现管理器 Actor（一个 UdpConnectedManager 类型的 Actor），然后绑定一个指定的 IP 和端口：

```
ActorRef udpManager = UdpConnected.get(getContext().system()).getManager();
udpManager.tell(UdpMessage.bind(getSelf(), new
InetSocketAddress("127.0.0.1",1998)), getSelf());
```

绑定成功后会收到 Udp.Bound 消息，此时我们不需要注册一个处理连接的 Actor。客户端在连接服务端时，需要发送 akka.io.UdpConnected.Connect 消息给 udpManager：

```
udpManager.tell(UdpConnectedMessage.connect(getSelf(), new
InetSocketAddress("127.0.0.1", 1998)), getSelf());
```

然后就可以直接向服务端发送数据了：

```
if (msg instanceof UdpConnected.Connected) {
    ActorRef conn=getSender();
    conn.tell(UdpConnectedMessage.send(ByteString.fromString("Hello Akka")),
getSelf());
}
```

## 6.7 Akka Streams

实际项目中往往离不开对数据的操作，这些数据可能来自很多地方，比如文件、网络或者数据库，我们一般把这些地方叫作数据源。在得到这些数据后，我们通常需要对它们做诸如提取、转换的处理，然后将它们输出到某个目的地，或者展示给用户。在这个过程中，我们需要考虑一个很重要的问题：当数据源源不断地从外部“流”进来时，我们该如何利用有限的资源高效地处理它们。

Akka 的 Streams 模块为解决这类问题提供了很好的思路。Akka Streams 建立在现有的 Actor 模型之上，它将所有的处理过程抽象成异步 & 并行执行的函数，而数据将会在这些函数内流动并得到处理，这种做法极大提高了整个计算的效率。假如我们把数据的流向简单地理解成“输入—输出”，那么在一个“流”里，可能存在多个“输入—输出”的操作，这些操作可以作为独立的组件而存在，在你需要它们时，将它们一个个装配起来即可。从这点可以看出，Akka Streams 为我们提供了一个简洁并易于扩展的编程模型。Akka Streams 另外一个特点是它可以让程序在有限资源（内存）下高效地处理大量数据，这其中离不开对内部缓冲池的有效控制，假如把数据的两端看作生产者和消费者，那么当数据的消费速度跟不上生产速度时，它就会减慢生产速度，反之亦然，这个行为被称为“背压”（BackPressure），它也是一个流量控制的手段。

### 6.7.1 Streams 组件

通过前面的介绍，想必大家对 Stream 已经有了初步的认识，那么我们该怎样去编写一个 Streams 程序呢？在动手写代码之前，我们先来了解一下 Akka Streams 的几个重要组件。

- Source：产生一个输出，它的下游在处理时会接收它的数据。
- Sink：需要一个输入，它通常是流处理的最后一个阶段。
- Flow：拥有一个输入和输出，它包含一些类似集合的操作，可以做一些数据转换、过滤等操作。
- RunnableGraph：当你拥有 Source、Sink 等结构后，还需要将它们连接在同一个管道上才能真正工作，此时我们需要构建一个 RunnableGraph 对象，为最后的执行（run）做好准备工作。

一个流总是由 Source 开始，它是流的起点，也是数据的来源，你可以从集合或者文件等数据源中构建一个 Source，然后将其经过 Flow（可以有多个）的转换或过滤，最后使用 Sink 来进行最后的处理。

下面我们先尝试一个简单的示例，以便让大家对 Stream 程序有个直观的认识，该示例将要实现的功能是：创建一个包含一系列数值的 Source，然后通过 Sink 循环打印出来。由于在 Akka 中，Streams 并不属于其核心模块，所以在使用它时需要先加入以下 maven 依赖：

```
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-stream-testkit_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
```

下面是这个例子的代码，我将在后面解释它的含义：

```
ActorSystem system = ActorSystem.create("sys");
Materializer materializer = ActorMaterializer.create(system);
Source<Integer, NotUsed> source = Source.range(1, 5);
Sink<Integer,CompletionStage<Done>> sink=Sink.foreach(System.out::println);
RunnableGraph<NotUsed> graph=source.to(sink);
graph.run(materializer);
```

我们先从 Source 看起，这里使用 Source.range 方法创建了一个包含 1 到 5 的整数的 Source 对象，作为数据的来源和流的起点。注意，现在仅仅使用它是无法将这些数据取出来的，也就是说，它目前只是描述了该怎样产生这些数据。Source 包括两个泛型：第一个泛型表示它产生的数据类型，第二个泛型表示运行时产生的其他辅助数据，假如没有则设置为 NotUsed。

为了能操作 Source 产生的数据，我们定义了一个 Sink 对象，它的 foreach 方法可以循环出 Source 中的每个元素，但是要达成这一步，需要调用 Source.to 方法将 Sink 连接起来，此时会创建 RunnableGraph 对象。到这里，流操作的准备工作才算正式完成。当执行 RunnableGraph.run 方法时，需要传入一个 Materializer 对象，它主要用来给流分配 Actor 并驱动其执行。

这个例子目前仅仅实现了基本的流处理，也没有引入 Flow 等组件的使用，尚不具备实际的用途。由于 Akka Streams 包含的内容较多且用法非常灵活，所以接下来我们先初步了解一下 Source、Sink、Flow 等组件的常见构建方式，然后再给出一个综合案例。

### 1. 构建 Source

1）从集合中构建 Source，并使用 runForeach 循环打印出来，注意：这里的 runForeach 实际上是对 runWith(Sink.foreach(f), materializer) 的包装。

```
List<String> list = new ArrayList<String>();
list.add("sh");
```

```
list.add("bj");
list.add("nj");
Source<String,NotUsed> s1=Source.from(list);
s1.runForeach(System.out::println, materializer);
```

2）从 Future 中构建 Source，比如通过 Patterns.ask 向 Actor 请求一个 Future 结果。

```
Source<String,NotUsed> s2=Source.fromFuture(Futures.successful("Hello
Akka!"));
```

3）构建可连续生产某类型数据的 Source，可以使用 limit 取出前 n 个数据。

```
// 重复的生成元素
Source<String, NotUsed> s3 = Source.repeat("Hello");
// 取出前5个
s3.limit(5).runForeach(System.out::println, materializer);
```

4）使用 FileIO API 从文件中构建 Source，可将文件内容作为流处理的输入。

```
Source<ByteString, CompletionStage<IOResult>> source =
FileIO.fromPath(Paths.get("demo_in.txt"));
```

### 2. 构建 Sink

1）使用 Sink 循环出每个元素。

```
Sink<Integer,CompletionStage<Done>> sink1=Sink.foreach(System.out::println);
```

2）使用 Sink 做 fold 运算，该运算会将 fold 的第一个参数作为初始值传入后面函数中的 x，每次计算后将结果继续作为下一次计算的参数输入。runWith 会将 Source 和 Sink 连接起来并运行。

```
Sink<Integer, CompletionStage<Integer>> sink2 = Sink.fold(1, (x,y) -> x*y);
CompletionStage<Integer> r1 = Source.range(1, 5).runWith(sink2,materializer);
r1.thenAccept(System.out::println);// 1*2*3*4*5=120
```

3）使用 Sink 做 reduce 运算，该运算和 fold 类似，不过没有初始参数。

```
Sink<Integer,CompletionStage<Integer>> sink3=Sink.reduce((x,y)->x+y);
CompletionStage<Integer> r2= Source.range(1, 5).runWith(sink3,materializer);
r2.thenAccept(System.out::println);// 1+2+3+4+5=15
```

4）使用 FileIO API 构建输出流的 Sink，结合 Source 可以实现简单的文件复制功能。

```
Sink<ByteString, CompletionStage<IOResult>> sink =
FileIO.toPath(Paths.get("demo_out.txt"));
```

### 3. 构建 Flow

Flow 需要一个输入和输出，一般作为中间过程而存在，在一个流操作中，我们可以包装多个 Flow 一起使用，图 6-2 展示了 Source、Flow、Sink 的搭配方式。

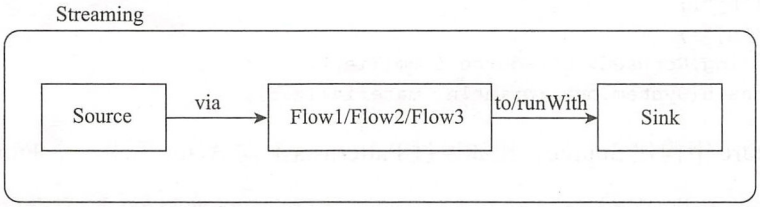



图 6-2 使用 Source、Flow、Source 来处理流数据

在使用时，Flow 组件通过调用 Source.via 方法附加在 Source 上，此时会构建一个新的 Source 对象，同时它也可以通过调用 Flow.to 方法附加在 Sink 上，此时会构建一个新的 Sink 对象。下面代码演示了这个过程：

```
Flow<String, Integer, NotUsed> flow = Flow.of(String.class).map(x -> {
    return Integer.parseInt(x) * 3;
});
Sink<Integer, CompletionStage<Done>> sink = Sink.foreach(System.out::println);
//list是一个包含5个元素的集合，代码略...
Source.from(list).via(flow).runWith(sink, materializer);
```

这段代码采用 via 方法将 Flow 组件附加在 Source 上，它实现的功能是：通过字符串集合构建出一个 Source 对象，然后将每个字符串取出来转换成数字并做相应计算，再使用 Sink.foreach 打印出来，其中转换的过程依赖于 Flow.map 方法来实现。如前面所说，我们也可以将 Flow 组件附加在 Sink 上以构建新的 Sink 组件，如：

```
Source.from(list).runWith(flow.to(sink), materializer);
```

通过这个例子还可以看出，流操作和集合非常相似，但是大家要注意，我们仍然不能把流等同于集合，它们之间有个很大的区别是：流的大小往往是未知的，无界的，而集合的大小一般都是已知的。所以在处理流时，你事先不需要知道要处理多少数据，生产者可能会源源不断地流入数据进来，流的底层能保证在有限的资源下处理“无限”的数据。所以实际上，它们的底层实现也是有很大区别的。

## 6.7.2 组合 Source、Sink

在前面的例子中，我们分别将 Flow 附加在 Source 和 Sink 上，由此创建了一个新的 Source 和 Sink 对象，实际上这个过程可以看成是一种“输入输出”的组合，如图 6-3 所示。

如图所示，由于 Source 是一个产生输出的组件，当它和 Flow 组合时，会生成一个拥有输出的 Source；同理，由于 Sink 是一个需要输入的组件，当它和 Flow 组合时，会生成一个需要输入的 Sink，这两种情况在前面已经实现过，这里不再给出示例。图中的第三种情况是 Sink 和 Source 组合，根据它们的结构特点，可以生成一个拥有输入输出的 Flow，下面是示例代码：

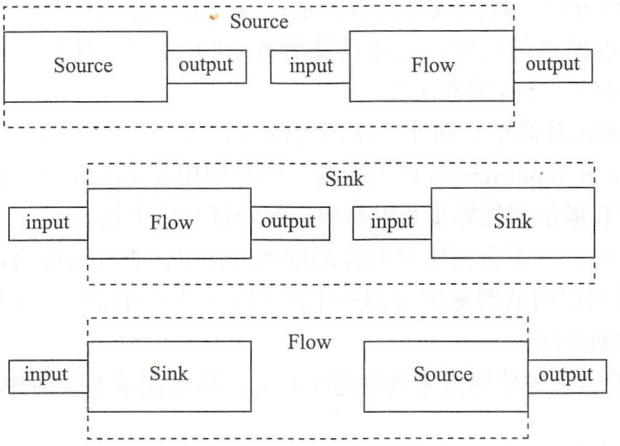



图 6-3 Flow、Source、Sink 组合

```
Flow<String, Integer, NotUsed> flow =
Flow.fromSinkAndSource(Sink.foreach(System.out::println), Source.range(1, 3));
```

实际上，还有一种常见的情况是多个 Flow 的组合，这通过 Flow.via 方法实现，很显然，这种组合会生成一个 Flow 组件。

### 6.7.3 案例：日志处理

在前面，我们已经了解了流的基本用法，但是并没有对它的处理过程做一个全面的分析。接下来，我们通过一个日志处理的案例来分析一下流的底层运行机制。

对于每个 Web 项目来讲，都会有专门的文件记录用户的访问信息，比如说 Tomcat 或者 Nginx 的 access_log，它们看起来如图 6-4 所示。

```
222.174.5.44 2015-08-10 GET /script 404
48.120.24.153 2015-08-11 GET /index.html 200
182.204.154.72 2015-08-11 GET /check_proxy.php 404
115.239.228.202 2015-08-11 GET /robots.txt 404
22.125.100.131 2015-08-14 GET /images/banner02.jpg 200
53.122.111.213 2015-08-15 GET /images/img1_1.png 200
56.151.32.123 2015-08-16 GET /images/tel.png 200
74.34.175.125 2015-08-17 GET /aboutus.html 200
83.216.55.200 2015-08-18 GET /contactus.html 200
77.111.23.42 2015-08-19 POST /index.html 404
56.151.32.123 2015-08-20 GET /images/tel.png 200
```

图 6-4 Web 项目中的 access_log

为了简化数据提取的难度，笔者已对原有的 access_log 做过调整。这个文件里面包含多条访问信息，每条访问信息由 IP、访问时间、访问方式、请求资源、状态这 5 个数据组

成，这些数据以空格分隔。我们现在要做的是：通过流的方式将其读取出来并封装成实体对象，并根据某些数据进行过滤，比如过滤所有 GET 请求，或者所有状态码为 200 的请求，然后将其写入另外一个待处理文件。

这个功能看起来比较简单，即使不使用流的方式去做，也可以很快编写出来，比如使用 Java I/O 中的 File 和 InputStream 即可实现。但是使用流来实现它，你将得到以下好处：

- 所有的读写和操作都是异步非阻塞的，吞吐量有较大提高；
- 在读取文件时，并不会把所有内容都读到内存里，否则可能会出现内存溢出。它的做法是根据数据的消费速度来调整生产（读文件）的速度，在其内部有一个可控的缓冲区来辅助处理。

接下来，我们看看怎样使用流来处理这个日志，该功能基本架构如图 6-5 所示。

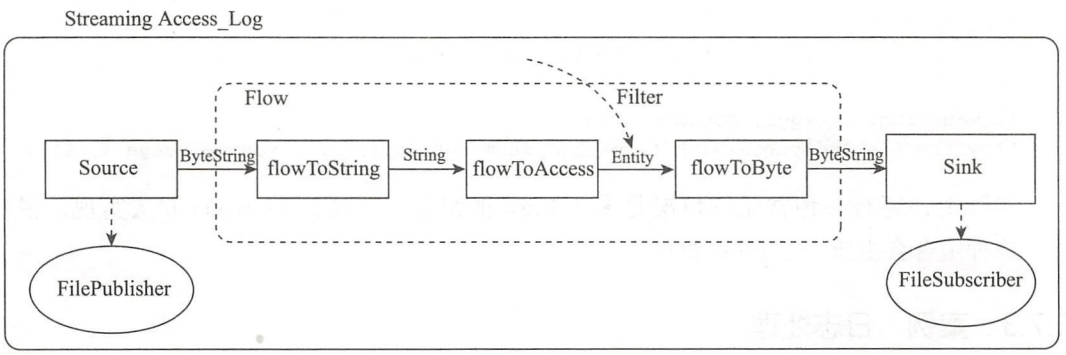



图 6-5 使用 Akka Stream 处理用户访问日志

该图展示了我们使用流来处理日志时需要用到哪些组件，以及数据将经历怎样的转换过程。首先，我们需要使用 FileIO API 创建一个“读”文件的 Source 对象，它产生的数据是 ByteString 而不是 String 类型的。由于我们需要将日志信息按每行来分隔，所以第一步要做的是将 ByteString 数据做一个转换，这个功能由 flowToString 这个 Flow 实现。当日志信息分隔成单独的字符串之后，我们希望结构化这些数据，将这些数据封装在一个实体对象里，以便后续程序更容易操作，这个过程实际上也是一个转换处理，该功能由 flowToAccess 这个 Flow 实现。当得到这些实体对象后，我们可以根据情况来过滤我们想要的数据，比如所有 404 的请求，那么此时可以对流进行 filter 处理。最后，我们需要将过滤的数据写入新的文件，由于在写入时需要使用 ByteString 类型的数据，所以这里使用一个 Flow 来进行转换，即 flowToByte。

在编码时，我们可以先实现“一头一尾”的功能，即 Source 和 Sink：

```
Source<ByteString, CompletionStage<IOResult>> source =
FileIO.fromPath(Paths.get("access_log.txt"));
Sink<ByteString, CompletionStage<IOResult>> sink =
FileIO.toPath(Paths.get("demo_out.txt"));
RunnableGraph<CompletionStage<IOResult>> graph = source.to(sink);
```



这里首先创建了一个 Source 对象，它是流处理的起点，也是数据的来源。由于需要从文件里读取内容，所以这里使用 FileIO 来创建 Source 对象。当通过 Source 输出文件后，需要使用 Sink 将文件输入到管道以进行后续处理。FileIO 底层使用了阻塞 I/O，在使用它创建 Source 和 Sink 时，底层会派发一个 Actor 在单独的 dispatcher 上，这个 dispatcher 可以在 akka.stream.blocking-io-dispatcher 进行配置。

source.to 方法会连接 Source 和 Sink，创建一个可运行的 graph。一旦这个 graph 被创建后，它就是不可变的。到这里所有准备工作已经就绪，假如要实现最终的效果，还需要调用其 run 方法，并传入一个 Materializer 对象，该对象主要用于将流执行于 Actor 之上，可以认为是流执行引擎。下面是执行 graph 的示例代码：

```
ActorSystem system = ActorSystem.create("sys");
Materializer materializer = ActorMaterializer.create(system);
CompletionStage<IOResult> coms = graph.run(materializer);
coms.thenAccept(System.out::println);
```

如代码所示，在 graph 运行时，它返回了一个 CompletionStage 对象，该对象描述了一个异步计算中的某个阶段，我们可以通过 thenAccept 或者 thenApply 等异步方法将里面包含的 IOResult 取出来。IOResult 主要用于描述一个 IO 操作的结果，它包含两个信息：count 和 status。count 表示从 Source 读取的字节数，status 表示结果状态，比如 Success（成功）或 Failure（失败）。在 Akka Stream 中，Source、Sink 等组件可以生成一些辅助性的数据，称为 Materialized Value，在我们这个示例中，Materialized Value 就是 CompletionStage<IOResult>。在运行这个程序时，底层会创建一组 Actor 来完成任务，graph 可以多次执行，每次执行都会产生一组新的 Actor。

这段代码比较简单，在 API 层面已经帮我们屏蔽了很多细节，不过我们可以稍微深入一下，看看它在底层会做哪些事情。

首先，在调用 FileIO.fromPath 方法时，底层会通过 FileSource 构造一个 Source 对象。在这个过程中，FileSource 会创建一个 FilePublisher 类型的 Actor，这个 Actor 在启动时（preStart）会打开文件通道对象 FileChannel。

然后，在调用 FileIO.toPath 方法时，底层会通过 FileSink 构造一个 Sink 对象。在这个过程中，FileSink 会创建一个 FileSubscriber 类型的 Actor，它在启动时会打开文件通道对象 FileChannel。

最后，在 RunnableGraph run 之后，FilePublisher 会从文件中读取数据（ByteString 类型），FileSubscriber 会将从 FilePublisher 接收到的数据写入文件。

在整个过程中，FilePublisher 不会将所有数据读到内存里，避免了 OutOfMemory 异常的产生，那么它是怎么读取数据的呢？ FileSubscriber 在写入之前会每次向 FilePublisher 请求一定的数据量，它不会大于 akka.stream.materializer.max-input-buffer-size 所设定的值，该值默认为 16。而 FilePulisher 也会依据请求的数据量来调整读取速度，所以 FilePublisher



读取的速度依赖于 FileSubscriber 写入的速度，不至于过快或过慢。这种做法可以使得 Stream 在有限的内存下处理无限的数据，内存方面我们是不用担心的。

回到刚才的示例，我们现在需要在 Source 和 Sink 之间搭建多个 Flow 组件来进行数据转换。由于 Source 在读取日志文件时，并不会自动区分每行数据，所以要先想办法将其分隔开来。这里我们可以借助 Framing 组件来实现：

```
Flow<ByteString, String, NotUsed> flowToString = Framing.
delimiter(ByteString.fromString("\r\n"), 100).map(x ->x.utf8String());
```

Framing.delimiter 方法可以创建一个用于处理非结构化字节流的 Flow 组件，这些字节流需要一个结尾的标记，以便能进行分割。日志文件都是以回车和换行符分隔的，这些字符在底层需要表现为某个 ByteString 类型的字节串，所以要提取每一行数据，需要将“\r\n”转换为 ByteString 类型的字节串，并将其传入 delimiter 方法，作为分隔的依据。delimiter 方法的第二个参数用于指定内部缓冲的最大长度，以便能找到匹配的分隔符，这样做能有效避免由于不合法的（数据）输入导致的内存溢出。在使用 delimiter 方法分隔后，数据仍然是 ByteString 类型的，而我们需要的是 String 类型的，所以在后面接着通过 map 方法将每行的 ByteString 数据转换成 String。

通过 flowToString 转换出来的数据仍然是一个非结构化的，比较“散”的数据，很多时候我们都希望将其封装在一个实体对象里，这样既提高了程序的可读性，也有利于其他程序的引用。要实现这个功能，我们也需要定义一个 Flow 来做转换处理：

```
Flow<String, AccessLog, NotUsed> flowToAccess = Flow.of(String.class).map(x ->
{
    String[] datas = x.split("");
    String ip = datas[0];
    String time = datas[1];
    String method = datas[2];
    String resource = datas[3];
    String state = datas[4];
    AccessLog access = new AccessLog(ip, time, method,resource, state);
    return access;
});
```

Flow.of 方法用于创建一个处理某类型数据的 Flow 组件，该类型由其参数指定，由于它上游的输出是 String，所以这里指定了 String 类型。此时 map 函数的参数就是从上游读取的每个字符串，在解析它们后，将每个信息封装在一个 AccessLog 实体对象里，AccessLog 只是一个普通的拥有 getter/setter 的实体类，具体代码不再列出。由于我们想把这些数据写入另外一个文件，而写入时需要转换为 ByteString 类型，所以我们定义了如下 Flow：

```
Flow<AccessLog, ByteString, NotUsed> flowToByte = Flow.
of(AccessLog.class).map(x -> ByteString.fromString(x.toString() + "\r\n"));
```



现在，我们已经定义好了三个 Flow 组件，只需要将它们装配起来就可以使用了：

```
RunnableGraph<CompletionStage<IOResult>> graph =
source.via(flowToString).via(flowToAccess).via(flowToByte).to(sink);
```

目前该程序会将所有访问信息都写入文件里，假如需要在写入之前做过滤，则可以使用 Flow.filter 实现，代码如下：

```
Flow<AccessLog,AccessLog,NotUsed> filter404=Flow.of(AccessLog.class)
.filter(x->x.getState().equals("404"));
```

这行代码过滤出了所有 404 的请求信息，你可以通过 via 方法将其装配在 flowToByte 前即可起作用，图 6-6 展示了该程序运行后的效果。

```
AccessLog [ip=222.174.5.44, time=2015-08-10, method=GET, resource=/script, state=404]
AccessLog [ip=182.204.154.72, time=2015-08-11, method=GET, resource=/check_proxy.php, state=404]
AccessLog [ip=115.239.228.202, time=2015-08-11, method=GET, resource=/robots.txt, state=404]
AccessLog [ip=77.111.23.42, time=2015-08-19, method=POST, resource=/index.html, state=404]
```

图 6-6 日志流程序运行效果（目标文件内容）

## 6.7.4 错误处理

当数据在各个函数间流动时，可能会由于某些原因产生一些错误，比如各种异常等，那么此时我们该如何处理这个流呢？

在默认情况下，当一个流出现异常时会马上停止，但是这可能并不是我们期望的。假如要改变这种行为，我们需要自定义 Supervision Strategy，即监督策略，这个概念和 Actor 中的“父监督”模式很类似。Supervision Strategy 为错误处理提供了以下几个可选项：

- Stop，停止当前流处理；
- Resume，丢弃错误的数据，继续运行；
- Restart，丢弃错误的数据，重新运行，之前的内部状态会清空。

以前面的日志处理为例，假如我们“故意”对日志信息做些处理，让它在 flowToAccess（见 6.7.3 节代码）处理时抛出 ArrayIndexOutOfBoundsException 异常，如图 6-7 所示。

```
222.174.5.44 2015-08-10 GET /script 404
48.120.24.153 2015-08-11 GET /index.html 200
182.204.154.72 2015-08-11 GET /check_proxy.php 404
115.239.228.2022015-08-11GET /robots.txt 404
22.125.100.131 2015-08-14 GET /images/banner02.jpg 200
53.122.111.213 2015-08-15 GET /images/img1_1.png 200
56.151.32.123 2015-08-16 GET /images/tel.png 200
74.34.175.125 2015-08-17 GET /aboutus.html 200
83.216.55.200 2015-08-18 GET /contactus.html 200
77.111.23.42 2015-08-19 POST /index.html 404
56.151.32.123 2015-08-20 GET /images/tel.png 200
```

图 6-7 错误的日志文件



假如此时直接运行该处理程序，它会抛出异常并终止流处理，由于异常是产生在第 4 行，所以前面两个 404 的信息仍然能够写入目标文件，如图 6-8 所示。

```
AccessLog [ip=222.174.5.44, time=2015-08-10, method=GET, resource=/script, state=404]
AccessLog [ip=182.204.154.72, time=2015-08-11, method=GET, resource=/check_proxy.php, state=404]
```

图 6-8　日志流程序抛出异常后的运行效果（目标文件内容）

同时大家可以自行观察一下 graph 在 run 之后的返回结果，它是一个 CompletionStage <IOResult> 类型的对象，该对象目前默认包含了读取数据的相关信息 (IOResult[count,status])，假如要观察写入数据的信息，则需要将 Source.to(sink) 方法改成 Source.toMat(sink,Keep.right()) 后再执行，此时大家会看到这样的结果：IOResult(185,Failure (java.lang.ArrayIndexOutOfBoundsException: 3))。Source.to 方法和 Source.toMat 方法功能类似，只不过默认保存左边（在我们的示例中就是读数据）的 Materialized Value，即 Source.to(sink)=Source.toMat(sink,Keep.left())。

回到示例，假如我们现在希望丢弃第 4 行并让程序继续运行下去，那么可以定义如下 Supervision Strategy：

```
Function<Throwable, Supervision.Directive> decider = err-> {
    if (err instanceof ArrayIndexOutOfBoundsException){
        return Supervision.resume();
    }else{
        return Supervision.stop();
    }};
```

大家会看到，我们可以让不同的异常对应不同的处理策略。上面这段代码的意思是：当流程序抛出 ArrayIndexOutOfBoundsException 时，让它丢弃错误数据并继续运行，即 Supervision.resume()；当抛出其他异常时，让它停止，即 Supervision.stop()。假如要让这个 Supervision Strategy 起作用，还需要在创建 Materializer 时将其配置进去，如下面代码所示：

```
Materializer materializer = ActorMaterializer.create(
ActorMaterializerSettings.create(system)
.withSupervisionStrategy(decider),system);
```

现在运行该程序，你会发觉错误的那条数据已经被丢弃，并且程序会继续处理后面的数据，最终效果如图 6-9 所示。

```
AccessLog [ip=222.174.5.44, time=2015-08-10, method=GET, resource=/script, state=404]
AccessLog [ip=182.204.154.72, time=2015-08-11, method=GET, resource=/check_proxy.php, state=404]
AccessLog [ip=77.111.23.42, time=2015-08-19, method=POST, resource=/index.html, state=404]
```

图 6-9　自定义 Supervision Strategy 后，日志流程序运行效果（目标文件内容）

假如大家希望出现异常后重新运行，可以将 Supervision.resume() 替换成 Supervision.restart()。



### 6.7.5 关联 Actor

Akka Streams 也是基于现有的 Actor 模型的，只不过在上层做了一些封装，以至于让我们感觉不到 Actor 的存在。实际上，我们可以将 Source、Sink 显式地和 Actor 关联起来使用：对于 Source 来讲，我们可以为它关联的 Actor 发送消息，这些消息会被发布到流中；对于 Sink 来讲，我们可以为它指定一个 Actor，进入 Sink 的数据将会被发送到这个 Actor 中。下面的代码演示了从 Source 中发送数据到 Sink 接收数据的完整过程：

```
Source<Object, NotUsed> source = Source.actorRef(10,
        OverflowStrategy.fail()).mapMaterializedValue(ref -> {
        ref.tell("AAA", ActorRef.noSender());
        ref.tell("BBB", ActorRef.noSender());
        ref.tell("CCC", ActorRef.noSender());
        ref.tell("DDD", ActorRef.noSender());
        return NotUsed.getInstance();
    });
    //SinkActor是一个普通的Actor，具体代码略
    ActorRef sinkActor =
system.actorOf(Props.create(SinkActor.class),"sinkActor");
    Sink<Object, NotUsed> sink = Sink.actorRef(sinkActor, "succ");
    source.runWith(sink, materializer);
```

其中，Source.actorRef 方法用于形成（materialized）一个 Actor，在调用这个方法时，需要传入以下两个参数：

- BufferSize：缓冲区的大小，当缓冲区不够用时，流操作会根据指定的策略来进行处理（比如丢弃某些元素或按失败处理）；
- OverflowStrategy：不能进入缓冲区时的处理策略。

以前面这段代码为例，我们将 BufferSize 设置为 10，当给其发送的消息超过这个缓冲值时，会根据 OverflowStrategy.fail() 来进行处理，此时 Actor 会接收到 akka.actor.Status.Failure 消息，该消息包含了相关异常信息。除了 OverflowStrategy.fail() 外，它还支持以下处理策略：

- OverflowStrategy.dropHead()：丢弃最“老”的元素；
- OverflowStrategy.dropTail()：丢弃最“年轻”的元素；
- OverflowStrategy.dropBuffer()：丢弃所有 buffer 中的元素；
- OverflowStrategy.dropNew()：丢弃新进入的元素。

当 Source 调用 mapMaterializedValue 方法后会将 Actor 以回调的方式取出来，然后我们就可以向其发送消息了，这些消息将被下游组件接收到。Sink 可以通过 actorRef 方法指定一个用于接收数据的 Actor，当任务成功完成后，会给这个 Actor 发送一个提示消息（即“succ”）。

运行该程序，大家就可以看到 SinkActor 已经接收到 Source 中发布的消息了。



## 6.8 本章小结

本章主要介绍了 Akka 提供的常用工具，它们在不同场景下都有广泛的应用，正确地使用这些工具不仅能提高开发效率，还能提高程序的扩展性及可维护性。在后面的第 7 章、第 8 章、第 9 章中，笔者将为大家介绍远程、分布式、HTTP 服务等高级主题。



第 7 章 Chapter 7

# 远　　程

一个复杂的业务通常会包含多个子模块，每个模块拥有单一的职责，并对外部提供服务。随着业务的发展，为了让系统不再受限于单台服务器的处理能力，这些模块可能会被拆分在不同服务器上（也可能只是不同 JVM），在 Akka 中，它们通常以 ActorSystem 的形式存在，那么我们怎么实现不同网络节点的 ActorSystem 之间的远程通信呢？实际上，Actor 模式本身已经包含了对远程或分布式网络节点的抽象，而且屏蔽了底层网络传输的细节，使得 ActorSystem 的远程通信和本地通信并没有明显的区别，当某个 Actor 给另外一个 Actor 发送消息时，并不需要知道对方是谁或来自哪里，你需要做的仅仅是：配置远程信息，获取到远程 Actor 的引用即可。当然，在实际项目中我们还需考虑一些细节问题，比如序列化，失败检测、远程事件等。在本章中，笔者将为大家全面介绍 Akka 的远程功能。

## 7.1　远程介绍

### 7.1.1　Java RPC

说到远程处理，大家可能会马上想到各种 RPC 框架。RPC（Remote Process Call），即远程过程调用，它的实现包括客户端和服务端，客户端发起请求后，服务端根据参数约定执行业务代码，然后将结果返回客户端，大致过程如图 7-1 所示。

实现 RPC 既可以基于 TCP 协议，也可以基于 HTTP 协议，它们各有优缺点。基于 TCP 协议的优点是，可以让开发者拥有更大的自主性，比如底层的协议定制及性能把控，但是会带来一定的复杂性。基于 HTTP 协议的优点是，已经有很多 Web 服务器（比如 Tomcat）帮你屏蔽了底层的实现细节，对开发者更加友好，但是也会带来一定的性能损耗（不过都可以接受）。



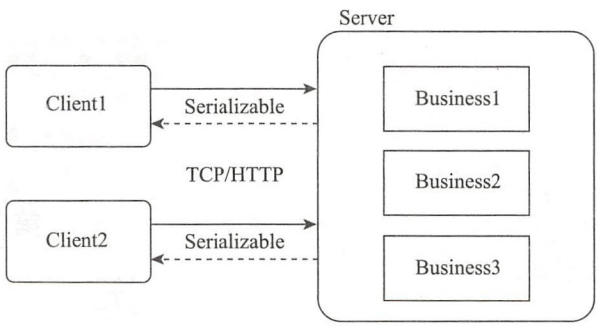



图 7-1 RPC 基本调用过程

在通过网络传输对象数据时，还需要考虑对象的序列化问题。对象序列化主要是指将对象转换成二进制流的过程，转换后的数据既可以保持在文件系统，也可以存储在内存中，并能通过网络进行传输。假如要通过二进制流重建为对象，则需要做反序列化处理。在 Java 中，实现序列化的第一步是将对象数据实现序列化接口，即：

```
class User implements Serializable{
    //其他代码略
}
```

然后可以通过 ObjectOutputStream 和 ObjectInputStream API 来序列化和反序列化，下面是部分核心代码：

```
User user=new User("1","afei");
//序列化到本地文件
ObjectOutputStream out=new ObjectOutputStream(new
FileOutputStream("user.ser"));
out.writeObject(user);
out.close();

//反序列化得到对象
ObjectInputStream in=new ObjectInputStream(new FileInputStream("user.ser"));
User userNew=(User)in.readObject();
in.close();
```

这段代码首先将 User 对象序列化到文件，然后反序列化重建该对象，实际上，我们还可以将该对象通过 RPC 的方式传输到远程。在 Java 中，使用 Socket API 就可以实现一个简单的 RPC 功能，下面看看具体做法。

首先，我们了解一下该 RPC 的大致实现思路，如图 7-2 所示。

对于一个不考虑高并发的 RPC 服务端来说，其中一个很重要的工作就是：定义请求参数，比如服务名称、方法名称、参数类型和参数值等，服务端正是通过这些参数来进行动态方法调用的。而客户端为了能实现透明的远程服务调用，需要通过代理的方式将 Socket 操作封装起来，这样客户端能达到和本地方法类似的调用体验。



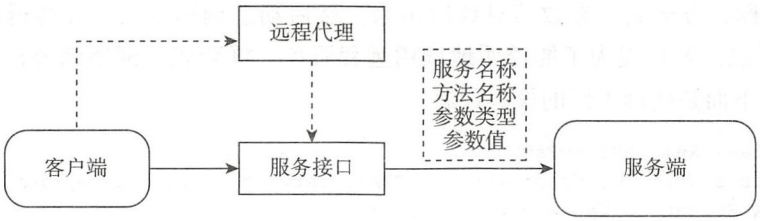



图 7-2 RPC 基本实现思路

现在，我们首先定义一个作为远程服务的接口和实现类：

```
public interface UserService {
    public String saveUser(User user);
}
public class UserServiceImpl {
    public String saveUser(User user) {
    System.out.println("新增用户: " + user);
    return "SUCCESS";
}}
```

然后定义服务端，并将服务注册起来。这里我们使用 ServerSocket 来创建服务，让其循环等待客户端的请求：

```
ServerSocket server = new ServerSocket(1988);
Map<String,Object> beanPool=new HashMap<String, Object>();
beanPool.put("org.rpcservice.service.UserService", new UserServiceImpl());
while (true) {
    //接收客户端请求
    Socket client = server.accept();
    ObjectInputStream in = new ObjectInputStream(
    client.getInputStream());
    //服务名称
    String serviceName=in.readUTF();
    //方法名
    String methodName=in.readUTF();
    //参数类型
    Class<?>[] paramType = (Class<?>[]) in.readObject();
    //参数
    Object[] params = (Object[]) in.readObject();
    //获取服务Service
    Object bean=beanPool.get(serviceName);
    Method method = bean.getClass().getMethod(methodName, paramType);
    Object result = method.invoke(bean, params);
    ObjectOutputStream out = new ObjectOutputStream(
    client.getOutputStream());
    out.writeObject(result);
    }
```

该服务端采用最基础的 ServerSocket 来实现，当接收到来自客户端的请求时，会把服



务（接口）名称、方法名、参数信息读取出来，然后动态调用方法，并将返回值发回客户端。如前面所说，客户端为了能透明地调用远程服务，屏蔽底层网络细节，可以为接口生成动态代理，下面是代理工厂的核心代码：

```
public class RemoteProxyFactory {
    public static Object createBean(Class service) throws Exception{
    final Socket socket = new Socket("127.0.0.1", 1988);
    Object serviceImpl=Proxy.newProxyInstance(service.getClassLoader(), new Class[]
        {service}, new InvocationHandler() {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
    out.writeUTF(service.getName());
    out.writeUTF(method.getName());
    out.writeObject(method.getParameterTypes());
    out.writeObject(args);
    ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
    Object result=in.readObject();
    return result;
    }});
    return serviceImpl;}}
```

当服务端启动后，需要先获取代理对象，才能调用远程方法：

```
UserService
    userService=(UserService)RemoteProxyFactory.createBean(UserService.class);
    String result=userService.saveUser(new User("123","afei"));
    System.out.println("result:"+result);
```

这几段代码仅仅简单实现了基于 TCP 协议的 RPC 功能，并不推荐将其直接使用在生产环境下。

### 7.1.2 Akka 远程

通过上面的 RPC 示例我们可以看到，当你实现远程调用时，是需要比较明确的区分“客户端 / 服务端”的，那么，Akka 的远程功能也是如此吗？实际上，Akka 的远程功能确实和 RPC 有类似之处，比如都需要借助 TCP 协议和序列化实现，但是 Akka 的远程是建立在对等网络的基础之上，即通信两端的地位都是一样的，不严格区分“客户端 / 服务端”，其中一端的 Actor 可以直接和另外一端的 Actor 产生连接并通信，不过在实际项目中，为了简化开发模型，我们可能会人为地区分出“消费者 / 服务者”两个角色 。

两个远程系统在通信时，它们的连接会经历四个状态的变化，即 Idle、Active、Gated、Quarantined，这些状态在不同条件下可以互相转换。在与一个远程系统进行通信之前，连接状态为 Idle ；当接收到一个远程系统的连接或尝试向远程发送消息时，连接状态变为 Active，此时两个系统之间可以正常收发消息；而当出现 TCP 连接失败、传输失败、远程

查找失败、远程系统停止等事件时，连接状态被设置为 Gated，此时所有对外的消息将会被丢弃，在一段时间后，该状态下的连接会重新变成 Idle，该时间可以使用 akka.remote.retry-gate-closed-for 进行设置，默认为 5s。在 Active 状态下，假如由于系统状态不一致导致的失败无法恢复，连接状态会变成 Quarantined，该状态是永久的，除非重启该系统。

在 Akka 远程中，每个 ActorSystem 都会运行在一个拥有 IP 和端口的节点之上，这些节点可能在同一个 JVM 上，也可能在不同 JVM 或者不同物理机上，节点的区分主要是通过 IP+ 端口共同来完成的，由于这些远程信息都是配置在 *.conf 文件里面的，所以在代码中并没有太多的体现，当然了，这个还与远程功能的使用方式有关。

Akka 远程主要提供三种方式供我们使用：

1）假如已知一个远程系统的 Actor，可以通过 ActorSystem 或 ActorContext 的 actorSelection 方法查询该 Actor，然后与其通信；

2）创建远程 Actor，每个 Akka 系统都应该包含该 Actor 的 class 文件；

3）创建基于 Pool 或 Group 的远程路由器。

下面我们将针对这三种方式，分别展开详细介绍。

## 7.2 创建远程 ActorSystem

要使用 Akka 的远程功能，需要先确保项目已经加入了 akka-remote 依赖，maven 配置如下：

```
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-remote_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
```

定义远程 ActorSystem 并不需要代码的支持，仅仅只需要在 *.conf 文件中做如下配置即可：

```
akka {
    actor {
    provider = "akka.remote.RemoteActorRefProvider"
    }

    remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
    hostname = "127.0.0.1"
    port = 2552
    }
  }
}
```

远程和本地的配置大部分都相似，区别主要在以下几个地方：

1）将 provider 配置成 akka.remote.RemoteActorRefProvider，表示提供远程功能。而默认的 provider 是 akka.actor.LocalActorRefProvider。

2）配置 akka.remote 节点，其中 enabled-transports 表示传输实现，我们可以通过 akka.remote.transport.Transport 接口来自定义该实现。hostname 和 port 分别表示服务的 IP 和端口号。假如不显式的配置 hostname，系统会默认调用 InetAddress.getLocalHost().getHostAddress() 设置该值。port 的默认值为 2552，假如设置为 0，系统会生成一个随机可用的端口。

按照上面的配置，一旦使用 ActorSystem.create 创建 ActorSystem 实例，将会在 127.0.0.1：2552 上启动一个监听，并打印如下日志：

```
[INFO] [07/29/2016 17:53:33.552] [main] [akka.remote.Remoting] Starting remoting
[INFO] [07/29/2016 17:53:33.756] [main] [akka.remote.Remoting] Remoting started;
    listening on addresses :[akka.tcp://sys@127.0.0.1:2552]
[INFO] [07/29/2016 17:53:33.762] [main] [akka.remote.Remoting] Remoting now
    listens on addresses: [akka.tcp://sys@127.0.0.1:2552]
```

此时该 ActorSystem 的远程访问地址是：akka.tcp://sys@127.0.0.1:2552，其中 sys 表示 ActorSystem 的名称。

## 7.3 Actor 远程访问

当一个远程 ActorSystem 启动后，我们可以通过远程地址直接访问到该系统下的 Actor。这个操作看起来是一个“客户端 - 服务端”类型的操作，但是实际上不是，在 Akka 中并没有直接提供专门的客户端 API 来访问服务端，服务的提供者和消费者必须都在 ActorSystem 环境下进行，它们的地位是对等的。

假如远程 ActorSystem 启动在 127.0.0.1:2552 节点上，并创建了一个名为“ rmtActor”的 Actor，那么该 Actor 的远程地址就是“ akka.tcp://sys@127.0.0.1:2552/user/rmtActor”，此时可以使用 actorSelection 方法获取该 Actor：

```
ActorSelection selection =
getContext().actorSelection("akka.tcp://sys@127.0.0.1:2552/user/rmtActor");
selection.tell("hello remoteActor", getSelf());
```

同之前讲的 RPC 一样，当向远程 Actor 发送对象数据时，该对象一定要能被序列化，最简单的方式是让该类型实现 java.io.Serializable 接口。

假如远程 Actor 由于某种原因停止了，那么直接通过 ActorSelection 发送消息肯定不会成功，并且 ActorSelection 本身是不能直接得到 Actor 引用的。此时比较好的做法是先向对方发送一个 Identify 信息，让对方自动返回一个 ActorIdentity 信息，该信息里就包含了 Actor 引用，然后就可以判断远程 Actor 是否可用：

仅供非商业用途或交流学习使用

```
ActorIdentity ai=(ActorIdentity)msg;
ActorRef targetRef=ai.getRef();
if(targetRef!=null){
    //...
}
```

当远程的 Actor 不可用时，在远程系统上做 DeathWatch 时和本地系统类似，即通过 getContext().watch 方法监控目标 Actor，当目标 Actor 终止或者由于网络原因不能正确地引用到 Actor 时，监控者就会接收到 Terminated 消息。这里大家应该注意到，Terminated 消息的产生与网络也是有关系的。在远程环境中，Akka 会通过发送心跳信息来检测远程节点是否可用，心跳间隔时间默认为 1s，该时间可以通过 akka.remote.watch-failure-detector.heartbeat-interval 来设置。但是这里有个问题：到底怎样的节点才会被判断为不可用？是通过超时设置来判断吗？实际上，Akka 在做远程故障检测时并没有简单地将超时作为判断依据。而是基于对历史心跳数据的统计，来计算出一个可信度的值，然后将这个值和 akka.remote.watch-failure-detector.threshold 设定的值做对比，假如达到这个值，则表示失败，在默认情况下 threshold 的值为 10，基本上可以满足大部分场景。在这个过程中，还有一个参数会影响检测效果，即 akka.remote.watch-failure-detector.acceptable-heartbeat-pause，它表示在发送心跳过程中的延迟，比如当本地系统和远程系统进行通信时突然断开网络，假如你不希望立马检测出故障，可以将该值设置大一点，当网络恢复后可以继续正常工作。总之，通过这种方式可以让故障检测变得更加智能，同时也避免了“一刀切”式的处理。

## 7.4 创建远程 Actor

在很多时候，远程 Actor 的创建会由“客户端”主导。当“客户端”通过配置或者代码指定远程创建节点时，它会通过网络将请求发送给该远程，然后委托它创建 Actor，此时客户端将拿到远程 Actor 的引用，后续使用方式和普通 Actor 并没有太大区别。假如要通过配置的方式指定节点，需要在 *.conf 下加上以下内容：

```
actor {
    provider = "akka.remote.RemoteActorRefProvider"
    deployment {
        /rmtCrtActor{
            remote = "akka.tcp://sys@127.0.0.1:2552"
    }
  }
}
#其他配置略...
```

在这段配置中，指定了 Actor 的名称为 rmtCrtActor，节点为 127.0.0.1:2552 上的 sys 系统。在创建时，务必要保证远程系统已经启动，并且远程和本地都要有 Actor Class，下面是部分测试代码：

仅供非商业用途或交流学习使用

```
ActorRef
ref=system.actorOf(Props.create(RmtCreateActor.class),"rmtCrtActor");
ref.tell("hello rmt",ActorRef.noSender());
```

注意，Actor 的名字一定要和配置中的路径对应。假如创建的是具有层级关系的 Actor，那么可以将路径配置为“/*/rmtCrtActor”。

假如在项目中，远程节点信息是需要动态获取的，那么可以考虑使用远程部署 API 来实现：

```
Address addr = new Address("akka.tcp", "sys", "127.0.0.1", 2552);
ActorRef ref=getContext().actorOf(Props.create(RmtCreateActor.class).
withDeploy(new Deploy(new RemoteScope(addr))));
```

## 7.5 远程路由器

使用路由器可以很方便地将消息按照一定的逻辑转发到目标 Actor，这对远程系统来讲也是适用的，下面笔者介绍一下远程路由的使用步骤。

假设现在有三个节点的系统，它们分别在 127.0.0.1:2701、127.0.0.1:2702、127.0.0.1:2703 上，我们要在 2703 上创建路由器，然后将消息路由到 2701 和 2702 的 Actor 中。为了让项目更加清晰，我们可以为三个节点分别创建配置文件，其中前两个节点的配置大致相似：

```
#remote1.conf, 其他配置省略
    remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
    hostname = "127.0.0.1"
    port = 2701
    }
}

#remote2.conf, 其他配置省略
remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
    hostname = "127.0.0.1"
    port = 2702
    }
}
```

2703 节点上需要配置路由策略和目标，假如要实现广播组路由，需要配置如下信息：

```
akka {
    actor {
    provider = "akka.remote.RemoteActorRefProvider"
```

仅供非商业用途或交流学习使用

```
    deployment{
        /rmtCommon {
            router = broadcast-group
            routees.paths = [
                "akka.tcp://sys@127.0.0.1:2701/user/rmt1",
                "akka.tcp://sys@127.0.0.1:2702/user/rmt2"]
            }
    }
}
remote {
enabled-transports = ["akka.remote.netty.tcp"]
netty.tcp {
    hostname = "127.0.0.1"
    port = 2703
  }
 }
}
```

假如是 group 类型的路由，需要将目标 Actor 的远程地址列表配置在 routees.paths 中；假如是 pool 路由，则需要将目标节点的远程地址配置在 target.nodes 中。

由于此时每个节点都有自己的配置，所以在代码中需要通过 ConfigFactory.load 方法手动加载配置文件，下面是这三个节点的核心代码：

```
//127.0.0.1:2701节点，RemoteActor1代码省略
ActorSystem
system=ActorSystem.create("sys",ConfigFactory.load("remote1.conf"));
system.actorOf(Props.create(RemoteActor1.class),"rmt1");

//127.0.0.1:2702节点，RemoteActor2代码省略
ActorSystem
system=ActorSystem.create("sys",ConfigFactory.load("remote2.conf"));
system.actorOf(Props.create(RemoteActor2.class),"rmt2");

//127.0.0.1:2703节点
ActorSystem system =
ActorSystem.create("sys",ConfigFactory.load("remote3.conf"));

//创建路由器
ActorRef router =
system.actorOf(FromConfig.getInstance().props(), "rmtCommon");

//路由消息
router.tell("hello rmt",ActorRef.noSender());
```

接着需要将其打成 jar 包，并通过 java 命令来启动该程序。在上面这个例子中，笔者将所有节点代码放在一个工程里，然后根据不同的配置文件来描述不同的节点信息，并且使用 maven 来打 jar 包，在运行时通过指定 Main-Class 来启动节点，这个过程看起来非常完美。但实际上，现在直接运行 jar 包是有问题的。因为 Akka 每个模块都有自己的配置文件

仅供非商业用途或交流学习使用

reference.conf，假如你直接打包，这些文件并不会自动合并到你的 jar 包中，这样就会导致一些默认信息的丢失。为了解决这个问题，需要在 pom.xml 文件中加入以下插件：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>1.5</version>
    <executions>
    <execution>
    <phase>package</phase>
<goals>
    <goal>shade</goal>
</goals>
    <configuration>
<shadedArtifactAttached>true</shadedArtifactAttached>
    <shadedClassifierName>allinone</shadedClassifierName>
    <artifactSet>
    <includes>
<include>*:*</include>
    </includes>
    </artifactSet>
    <transformers>
    <transformer
    implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
    <resource>reference.conf</resource>
    </transformer>
    <transformer
    implementation="org.apache.maven.plugins.shade.resource.ManifestResource
        Transformer">
    <manifestEntries>
    <Main-Class>com.akka.remote3.Remote3</Main-Class>
</manifestEntries>
</transformer>
</transformers>
</configuration>
</execution>
</executions>
</plugin>
```

此时在使用 maven 命令打包后，可以看到 target 目录下会生成一个 allinone 的 jar 包，现在，你可以使用下面命令来启动各个节点了：

java -cp RemoteTest-1.0-allinone.jar  org.main.RemoteMain1

java -cp RemoteTest-1.0-allinone.jar  org.main.RemoteMain2

java -cp RemoteTest-1.0-allinone.jar  org.main.RemoteMain3

2701、2702、2703 分别启动后的效果如图 7-3 所示。

仅供非商业用途或交流学习使用

```
[root@master test]# java -cp RemoteTest-1.0-allinone.jar  org.main.RemoteMain1
[INFO] [11/18/2017 17:17:36.152] [main] [akka.remote.Remoting] Starting remoting
[INFO] [11/18/2017 17:17:36.286] [main] [akka.remote.Remoting] Remoting started; listening on addresses :[akka.tcp://sys@127.0.0.1:2701]
[INFO] [11/18/2017 17:17:36.288] [main] [akka.remote.Remoting] Remoting now listens on addresses: [akka.tcp://sys@127.0.0.1:2701]
Actor[akka://sys/user/rmt1#698901315]:hello rmt from Actor[akka.tcp://sys@127.0.0.1:2703/deadLetters]
```

```
[root@master test]# java -cp RemoteTest-1.0-allinone.jar  org.main.RemoteMain2
[INFO] [11/18/2017 17:17:38.748] [main] [akka.remote.Remoting] Starting remoting
[INFO] [11/18/2017 17:17:38.875] [main] [akka.remote.Remoting] Remoting started; listening on addresses :[akka.tcp://sys@127.0.0.1:2702]
[INFO] [11/18/2017 17:17:38.876] [main] [akka.remote.Remoting] Remoting now listens on addresses: [akka.tcp://sys@127.0.0.1:2702]
Actor[akka://sys/user/rmt2#-1840929365]:hello rmt from Actor[akka.tcp://sys@127.0.0.1:2703/deadLetters]
```

```
[root@master test]# java -cp RemoteTest-1.0-allinone.jar  org.main.RemoteMain3
[INFO] [11/18/2017 17:17:42.827] [main] [akka.remote.Remoting] Starting remoting
[INFO] [11/18/2017 17:17:42.956] [main] [akka.remote.Remoting] Remoting started; listening on addresses :[akka.tcp://sys@127.0.0.1:2703]
[INFO] [11/18/2017 17:17:42.957] [main] [akka.remote.Remoting] Remoting now listens on addresses: [akka.tcp://sys@127.0.0.1:2703]
```

图 7-3 远程路由器运行效果

## 7.6 远程事件

通过远程事件，我们可以及时了解远程系统的状态，比如是否成功连接、断开连接、出现故障等。假如要订阅与远程相关的任何事件，可以使用 RemotingLifecycleEvent，它是远程的顶级事件。由于 EventStream 实现了 SubchannelClassification 分类器，所以可以监听到它的任何子级事件。表 7-1 列举了它的部分子级事件。

表 7-1 RemotingLifecycleEvent 子事件

| 事 件 | 作 用 |
| --- | --- |
| akka.remote.AssociatedEvent | 与远程系统连接成功时，将触发该事件，事件信息中包含本地、远程的地址 |
| akka.remote.DisassociatedEvent | 与远程系统断开连接时，将触发该事件，事件信息中包含本地、远程的地址 |
| akka.remote.AssociationErrorEvent | 当连接出现错误时，将触发该事件，事件信息除了包含本地、远程的地址外，还包括错误信息 |
| akka.remote.RemotingListenEvent | 当远程系统启动并监听端口时，将触发该事件，此时会在控制台打印“Remoting now listens on addresses...” |
| akka.remote.RemotingShutdownEvent | 当远程系统 shutdown 时，将触发该事件，此时会在控制台打印“Remoting shut down” |
| akka.remote.ThisActorSystemQuarantinedEvent | 当前系统被远程系统隔离时，将触发该事件 |

假如要使用 EventStream 订阅事件，需要先创建订阅 Actor，然后调用 EventStream.subscribe 方法，并将事件传入：

```
system.eventStream().subscribe(eventActor, AssociationErrorEvent.class);
```

当事件被触发时，eventActor 就会收到对应的事件对象：

```
if(msg instanceof AssociationErrorEvent){
AssociationErrorEvent aee=(AssociationErrorEvent)msg;
```

仅供非商业用途或交流学习使用

```
System.out.println(aee.getLocalAddress()+"-->"+aee.getRemoteAddress()+":"+ae
e.getCause());
}
```

## 7.7 序列化

只要涉及远程传输，必然会用到序列化功能。在 7.1.1 节，笔者使用 ObjectOutputStream 和 ObjectInputStream API 实现了一个简易的序列化功能，它要求对象类型必须实现 java.io.Serializable 接口，否则会序列化失败。在 Akka 远程中，默认也会采用 Java 的序列化方式，在 *.conf 文件中，序列化接口和序列化实现按照如下方式配置：

```
akka{
    actor {
        serializers {
            java = "akka.serialization.JavaSerializer"
            #其他配置略...
        }
        serialization-bindings {
            "java.io.Serializable" = java
            #其他配置略...
        }
    }
}
```

该配置的意思是，实现了 java.io.Serializable 接口的消息类型都会通过 akka.serialization.JavaSerializer 进行序列化，JavaSerializer 的底层依赖于 ByteArrayOutputStream 和 ObjectOutputStream 等 Java API 来实现，但是考虑到 Java 原生序列化的性能问题，我们可能还需要借助其他序列化组件的支持。

### 7.7.1 Protobuf 框架

Protobuf 是 Google 的一款高性能的序列化框架，它拥有平台无关性，并提供多语言支持，是远程及分布式系统最为常用的序列化工具之一。在 Akka Remote 模块中，提供了 akka.remote.serialization.ProtobufSerializer 组件，可以让 akka.protobuf.GeneratedMessage 和 com.google.protobuf.GeneratedMessage 类型的消息支持 Protobuf 序列化。前者的所有子类型，都可以在程序里面直接使用，而后者需要先确保工程已经加入了 Protobuf 的依赖。

假如要直接使用 Google Probuf，需要先在 https://github.com/google/protobuf/releases 下载 protoc-2.5.0-win32.zip，然后解压，会看到目录下有一个 protoc.exe 文件，它可以用于生成消息源文件。

接下来，我们需要创建一个数据描述文件，取名 UserProEntity.proto，它的内容如下：

```
package akkajava;
```

仅供非商业用途或交流学习使用

```
option java_package="com.akka.rmt";
option java_outer_classname="UserProEntityTran";
message UserProEntity{
    required string username=1;
    required string address=2;
}
```

其中，java_package 指定生成的 Java 文件所在的 package，java_outer_classname 指定 class 名称，而真正的消息就是 UserProEntity，它拥有两个字符串的字段。接着使用 protoc.exe 生成 Java 文件，命令如下：

```
protoc.exe --java_out=akkatest UserProEntity.proto
```

其中，--java_out 参数指定输出文件的位置。执行完该命令，就可以看到在该目录的 package 下生成了一个名为 UserProEntityTran 的 Java 文件。接着将其拷贝到项目中，会发现该类下已经引用了 Protobuf 的代码，所以必须在 maven 中加入以下依赖：

```
<dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>2.5.0</version>
</dependency>
```

打开 UserProEntityTran.UserProEntity 这个类，会看到它确实继承了 com.google.protobuf.GeneratedMessage，那么在传输这个消息数据时，就会使用 ProtobufSerializer 来进行序列化，下面是 UserProEntity 消息的构建示例：

```
//创建构建工具
UserProEntityTran.UserProEntity.Builder
builder=UserProEntityTran.UserProEntity.newBuilder();
//设置内容
builder.setUsername("yunfei--afei");
builder.setAddress("sh");

//构建消息
UserProEntity upe=builder.build();

//传输到远程
selection.tell(upe, getSelf());
```

假如大家使用的是 protoc-3.x 的版本，那么生成的消息会继承 com.google.protobuf.GeneratedMessageV3，此时并不会匹配到 ProtobufSerializer，默认会使用 JavaSerializer 来进行序列化，假如希望使用 ProtobufSerializer，则需要在 serialization-bindings 配置 "com.google.protobuf.GeneratedMessageV3"=proto。

## 7.7.2 序列化 API

在 Akka 中，序列化组件是以扩展的形式存在的，该扩展用 Serialization 表示。在一个

仅供非商业用途或交流学习使用

系统中，可能配置了多个序列化实现，如果想知道某个消息类型到底使用了什么工具进行序列化，可以调用 Serialization.findSerializerFor 方法来进行查找，假如找不到则抛出异常。一旦得到序列化工具 Serializer，就可以分别调用它的 toBinary 和 fromBinary 方法来进行序列化和反序列化，下面是一个简单示例：

```
//得到序列化扩展
    Serialization serialization = SerializationExtension.get(system);?

    //新建消息
    UserProEntityTran.UserProEntity.Builder
builderUserPro=UserProEntityTran.UserProEntity.newBuilder();
    builderUserPro.setUsername("yunfei");
    builderUserPro.setAddress("sh");
    UserProEntity upe=builderUserPro.build();

    //得到序列化工具
    Serializer serializerUserPro = serialization.findSerializerFor(upe);

    System.out.println("使用序列化组件:  "+serializerUserPro);

    //序列化到字节数组
    byte[] bys=serializerUserPro.toBinary(upe);

    //反序列化成对象
    UserProEntity upedemo=(UserProEntity)serializerUserPro.
fromBinary(bys,UserProEntity.class);

    System.out.println(upedemo.getUsername()+"-->"+upedemo.getAddress());
```

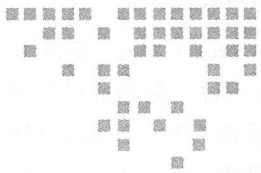
假如你希望自定义序列化实现，可以继承 JSerializer 这个抽象类，然后将其配置在 *conf 文件的 serializers、serialization-bindings 节点中。

## 7.8 本章小结

得益于 Akka 框架的良好设计，使得其远程操作和本地操作并没有太大区别，这在很大程度上简化了远程应用的开发。通过 akka-remote 模块，我们大致有三种方式使用远程功能：查询并访问已存在的远程 Actor；创建并使用远程 Actor；创建并使用远程路由。当应用涉及网络通信时，出现故障的可能性肯定会增大，为了及时了解当前网络及通信的状态，我们需要订阅相关事件，比如 AssociatedEvent、DisassociatedEvent、AssociationErrorEvent 等。在网络应用中，一个影响其性能的关键因素是：序列化。Java 提供的默认序列化功能性能不佳，我们可以有选择地使用其他序列化框架，比如 Protobuf，或者你也可以自定义序列化实现。

远程是 Akka 非常好用的高级功能之一，也是集群的基础。在下一章中，笔者将全面讲解 Akka 集群的搭建和应用。

仅供非商业用途或交流学习使用



第 8 章 Chapter 8

# 集　群

随着业务的高速发展，单台服务器可能无法承载日益增大的请求压力，在这种情况下，我们可以对该系统做水平扩展，让多个节点同时服务，以此突破单机处理能力、容灾能力的瓶颈，保证系统的高性能及高可用，这也就是我们常说的分布式集群。得益于 Actor 模型的良好实现，Akka 天生拥有分布式的能力，从第 7 章所讲的“远程”功能可以看出，操作一个远程 Actor 的方式和操作本地 Actor 并没有明显的区别，Akka 基本上已经帮我们屏蔽了底层的网络通信细节。Akka 的集群功能由 akka-cluster 模块提供，它基于去中心化的 P2P 模型，没有单点故障和单点瓶颈，可以满足多个应用场景。在本章中，笔者将为大家介绍 Akka 集群的常规用法。

## 8.1　Akka 集群概述

Akka 集群由一系列节点（node）组成，这些节点分布在不同 JVM 或者物理机上，每个节点都使用 hostname:port:uid 来唯一标识。当集群系统启动时，每个种子节点（seed-node）将会自动加入该集群，此时其他节点可以发送 join 命令给这些种子节点，要求加入该集群。在集群运行阶段，它的节点也并不固定，我们可以根据业务情况动态地加入新的节点，或者停掉某个节点，在为整个集群服务的过程中，节点本身也会经历“生老病死”的阶段，我们称之为生命周期。通常来讲，处在集群中的任何一个节点都可能会产生故障，从而导致该节点上的服务不可用，此时集群应该及时探测到失败节点并做相应处理，然后将最新的状态同步到其他健康节点。在理想情况下，任何节点状态的变化都会通过 gossip 协议传输到整个集群网络的每个节点，以此保证集群状态的一致性。

### 8.1.1 种子节点

种子节点（seed-nodes）和普通节点并没有太大区别，只不过它们被配置在 cluster.seed-nodes 中，作为一种“创始成员”而存在，在一个集群中，可以配置多个种子节点。当集群启动时，不要求所有种子节点都启动，但是排在 seed-nodes 列表中的第一个节点必须启动，否则其他节点没法 join 进集群。当新的节点启动时，它会发送消息给所有的 seed nodes，然后发送 join 命令给第一个应答节点，假如没有任何一个节点应答，它会继续重试，直到成功或者停止。种子节点的配置如下所示：

```
seed-nodes = [
    "akka.tcp://sys@127.0.0.1:2551",
    "akka.tcp://sys@127.0.0.1:2552"]
```

### 8.1.2 领导节点

领导节点（Leader Node）主要负责管理集群中成员节点的状态，比如说将一个 joining 状态下的节点设置为 up 状态，或者将一个 exiting 状态下的节点设置为 removed 状态。在集群中，任何节点都可以成为 leader。

### 8.1.3 节点状态及生命周期

在节点的生命周期中，节点的状态会发生一些改变，在常规情况下，节点有如下状态，状态变化如图 8-1 所示。

- joining，加入集群时的临时状态；
- up，正常操作状态；
- leaving/exiting：正常移出状态；
- down：不再是集群能选择的节点；
- removed：彻底移除，不再是集群成员。

如图 8-1 所示，在最开始，节点处于 joining 状态，当这个状态被所有其他节点同步到之后，Leader 节点会将其设置为 up 状态，此时它可以正常地为集群服务。当节点以安全的方式离开集群时，Leader 节点会将其设置为 leaving 状态，当这个状态被所有其他节点同步之后，Leader 节点会将其设置为 exiting 状态。当 exiting 状态又被其他节点同步到之后，Leader 节点会将其设置为 removed 状态，并将其从集群中移除。

当某个节点由于意外（比如直接停止）而被集群认为是 unreachable 时，假如有新的节点想加入集群，Leader 节点是不会将其设置为 up 的（但依然能收到 join 请求），除非 unreachable 节点重新加入或者被设置为 down。当它重新加入集群时，Leader 节点会将其先 remove 掉，在其他节点都同步到它的 joining 状态时，Leader 节点会再次将其设置为 up 状态。

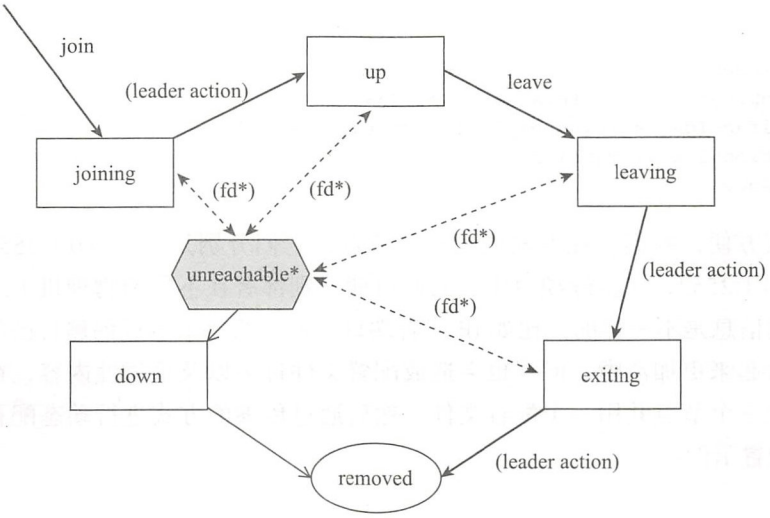



图 8-1 集群节点生命周期（状态变化，图片来自官网）

### 8.1.4 故障检测

在集群中，某个节点出现故障的可能性还是比较大的，我们需要系统能及时并准确地检测出这些故障。检测某个节点是否可用的第一步是向其发送心跳信息并期望对方给出正确反馈，假如不能，则说明对方有可能已经不可达（unreachable），为了降低整个集群中探测节点状态时的开销，每个节点都会被一定数量的其他节点监视，而不是被所有节点监视，这个数量可以通过 akka.cluster.failure-detector.monitored-by-nr-of-members 进行设置，默认情况下，这个值为 5。在检测某个节点是否是 unreachable 时，集群中的故障检测器会通过历史故障统计和心跳信息计算出一个故障可信度的值 phi，然后根据该值和 akka.cluster.failure-detector.threshold 做对比，当 phi<threshold 时表示可用，反之不可用，这种判断方式和远程一样。一旦确认某个节点确实是 unreachable 时，该状态信息会通过 gossip 协议传播到整个集群。

## 8.2 创建 Akka 集群

使用 Akka 创建一个基础集群几乎不需要代码的参与，我们要做的仅仅是将本地配置换成集群配置，然后同时启动多个节点即可。在本节中，笔者将创建一个拥有三个节点的集群，该集群实现的功能非常简单：当其中的某个节点发生状态改变时，集群中其他节点能收到该信息，该功能几乎不涉及任何真实业务。

### 8.2.1 代码及集群配置

在将项目配置成集群环境之前，务必保证项目中已经加入了集群模块的依赖，maven

配置如下：

```
    <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-cluster_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
```

为了测试方便，我们将在本地创建三个节点，它们分别是 :127.0.0.1:2550、127.0.0.1:2551、127.0.0.1:2552，在实际项目中，它们可能分别部署在不同的物理机上。按道理，每个节点的配置信息是不一样的，比如 IP 或者端口，那么给每个节点创建自己的配置文件会让程序结构看起来更加清晰，但是也会造成配置文件过多以及重复性内容，在我们这个示例中，可以让三个节点共用一个配置文件，然后通过传参的方式进行动态配置。下面是一个最基础的配置示例：

```
akka {
    #loglevel=OFF
    actor {
    provider = "akka.cluster.ClusterActorRefProvider"
    }
    remote {
    log-remote-lifecycle-events = off
    netty.tcp {
    hostname = "127.0.0.1"
    port = 2550
    }
  }
cluster {
    seed-nodes = [
    "akka.tcp://sys@127.0.0.1:2551",
    "akka.tcp://sys@127.0.0.1:2552"]
  }
}
```

**配置说明**

1）将 provider 从 akka.actor.LocalActorRefProvider 换成 akka.cluster.ClusterActorRefProvider，表示当前系统处于集群环境；

2）定义 remote 中的 hostname 和 port，即当前系统启动的节点信息；

3）seed-nodes 配置了当前集群的种子节点，集群并不要求这些种子节点都能同时启动，而且它们的启动顺序也是不固定的。

假如现在要共用这套配置，很明显需要在代码中动态设置 hostname 和 port，为了在集群启动后动态地加入新的节点，我们可以将 port 设置为启动参数。

当一个集群启动后，每个节点的状态都会被传播到其他节点，假如要得到这些信息，我们可以订阅相关事件，下面是一个简单示例：

```
public class ClusterDemo extends UntypedActor {
    Cluster cluster = Cluster.get(getContext().system());

    @Override
    public void preStart() {
        cluster.subscribe(getSelf(), ClusterEvent.initialStateAsEvents(),
        UnreachableMember.class,MemberEvent.class);
    }

    @Override
    public void postStop() {
        cluster.unsubscribe(getSelf());
    }

    @Override
    public void onReceive(Object message) {

    if (message instanceof MemberUp) {
        MemberUp mUp = (MemberUp) message;
        Member m=mUp.member();
        System.out.println(getSelf()+"-->Member "+m+"is Up: the role is "+m.roles());
    } else if (message instanceof UnreachableMember) {
        UnreachableMember mUnreachable = (UnreachableMember) message;
        Member m=mUnreachable.member();
        System.out.println(getSelf()+"-->Member "+m+" detected as unreachable: the
role is "+m.roles());
    } else if (message instanceof MemberRemoved) {
        MemberRemoved mRemoved = (MemberRemoved) message;
        Member m=mRemoved.member();
        System.out.println(getSelf()+"-->Member "+m+" is Removed: the role is"+m.
            roles());
    } else if (message instanceof MemberEvent) {
        MemberEvent me=(MemberEvent)message;
        Member m=me.member();
        System.out.println(getSelf()+"-->MemverEvent: "+me+""+m.roles());
    } else {
        System.out.println(getSelf()+"-->Other: "+message);
        unhandled(message);
    }
    }
}
```

在这个 Actor 中，首先通过 Cluster.get(getContext().system()) 方法得到当前集群对象，然后调用 subscribe 方法让当前 Actor 订阅 UnreachableMember、MemberEvent 事件，其中 UnreachableMember 事件会在某个节点被故障检测器 (failure detector) 认定为不可达 (unreachable) 时触发，MemberEvent 事件是成员状态的顶级事件，它的子级事件及说明如表 8-1 所示。

表 8-1 集群成员状态事件

| 事件对象 | 说 明 |
| --- | --- |
| MemberJoined | 某节点加入集群，状态为 joining 时触发该事件 |
| MemberUp | 当 Leader 节点将某节点设置为 up 状态时触发该事件 |
| MemberLeft | 当某节点离开集群时，Leader 节点会将其设置为 leaving 状态，此时触发该事件 |
| MemberExited | 当所有节点都同步到某节点的 leaving 状态时，Leader 节点会将其设置为 exiting 状态，此时触发该事件 |
| MemberRemoved | 当某节点被移除后触发该事件 |

在 onReceive 方法中，通过匹配不同事件来了解节点状态信息，从这些事件中我们都能得到一个很重要的对象，即 Member，该对象拥有当前节点的地址信息、状态信息、角色信息。实际上在这里是不需要精确匹配 MemberUp、MemberRemoved 类型的，因为它们都是 MemberEvent 的子类，而后者已经能覆盖所有状态信息了，但是在实际项目中，大部分情况下是需要对每个状态做精确判断的。

## 8.2.2 启动集群

为了能够动态地启动任何端口的节点，我们可以将端口设置为 main 方法的参数，然后通过 ConfigFactory 对象动态配置 akka.remote.netty.tcp.port 信息：

```
public static void main(String[] args) {
    String port = args[0];
    String sysname=args[1];
    Config config = ConfigFactory.parseString("akka.remote.netty.tcp.port="+ port).
        withFallback(ConfigFactory.load());
    ActorSystem system = ActorSystem.create(sysname, config);
    system.actorOf(Props.create(ClusterDemo.class),"clusterDemo"+port);
}
```

然后我们将系统打成 jar 包，并通过 Java 命令执行，在使用 maven 打包前，应该要在 pom.xml 中加入 maven-shade-plugin 插件，大家可以参考 7.5 节中的配置。图 8-2、图 8-3、图 8-4 分别展示了 2550、2551、2552 的启动过程。

```
[root@AY121205084402322253l opt]# java -cp AkkaClusterDemo-1.0-allinone.jar com.akka.book.StartUtils 2550 sys
Actor[akka://sys/user/clusterDemo2550#953682420]-->MemverEvent: MemberJoined(Member(address = akka.tcp://sys@127.0.0.1:2550, status = Joining)) Set()
Actor[akka://sys/user/clusterDemo2550#953682420]-->Member Member(address = akka.tcp://sys@127.0.0.1:2551, status = Up)is Up: the role is Set()
Actor[akka://sys/user/clusterDemo2550#953682420]-->Member Member(address = akka.tcp://sys@127.0.0.1:2550, status = Up)is Up: the role is Set()
Actor[akka://sys/user/clusterDemo2550#953682420]-->MemverEvent: MemberJoined(Member(address = akka.tcp://sys@127.0.0.1:2552, status = Joining)) Set()
Actor[akka://sys/user/clusterDemo2550#953682420]-->Member Member(address = akka.tcp://sys@127.0.0.1:2552, status = Up)is Up: the role is Set()
```

图 8-2 127.0.0.1:2550 节点启动信息

```
[root@AY121205084402322253l opt]# java -cp AkkaClusterDemo-1.0-allinone.jar com.akka.book.StartUtils 2551 sys
Actor[akka://sys/user/clusterDemo2551#1700726542]-->Member Member(address = akka.tcp://sys@127.0.0.1:2551, status = Up)is Up: the role is Set()
Actor[akka://sys/user/clusterDemo2551#1700726542]-->MemverEvent: MemberJoined(Member(address = akka.tcp://sys@127.0.0.1:2550, status = Joining)) Set()
Actor[akka://sys/user/clusterDemo2551#1700726542]-->Member Member(address = akka.tcp://sys@127.0.0.1:2550, status = Up)is Up: the role is Set()
Actor[akka://sys/user/clusterDemo2551#1700726542]-->MemverEvent: MemberJoined(Member(address = akka.tcp://sys@127.0.0.1:2552, status = Joining)) Set()
Actor[akka://sys/user/clusterDemo2551#1700726542]-->Member Member(address = akka.tcp://sys@127.0.0.1:2552, status = Up)is Up: the role is Set()
```

图 8-3 127.0.0.1:2551 节点启动信息

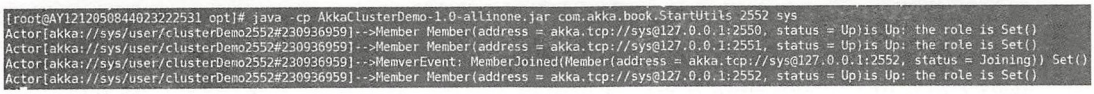



图 8-4 127.0.0.1:2552 节点启动信息

由于在集群启动时会产生大量的日志，不利于我们观察自己输出的信息，所以笔者在这里已将 akka.loglevel 设置为 OFF。从截图显示的信息来看，当某个节点启动后，它会加入到集群中，当节点状态变为 up 时，所有其他节点都能收到 MemberUp 信息。假如此时停掉某个节点，其他节点也是能够收到 UnreachableMember 信息的。最后要注意的是，集群中所有成员的 ActorSystem 名称必须一样。

### 8.2.3 加入集群

在 8.1.1 节介绍过，当新的节点启动时，它会发送消息给所有的 seed-nodes，然后发送 join 命令给第一个应答节点，实际上，新节点也可以给任何一个集群中的节点发送 join 命令，不必非得是 seed-nodes。当 seed-node 为空时，系统启动时不会加入任何节点，我们可以调用 Cluster.join 方法将当前节点加入指定的 Address，示例代码如下：

```
Config config =
ConfigFactory.parseString("akka.remote.netty.tcp.port=2555")
.withFallback(ConfigFactory.load("appcluster.conf"));
    ActorSystem system = ActorSystem.create("sys", config);
    system.actorOf(Props.create(ClusterDemo.class),"scListener_JoinDemo");
    Cluster cluster = Cluster.get(system);
    Address address = new Address("akka.tcp", "sys", "127.0.0.1",2551);
    cluster.join(address);
```

其中 appcluster.conf 中的 seed-nodes 已经设置为空数组 []，其他配置和前面的示例完全相同，这里不再给出。当执行上面这段代码后，当前节点会向 127.0.0.1:2551 节点发送 join 请求并获得对方的欢迎提醒，此时它处于 joining 状态，当这个状态传播到整个集群之后，Leader 节点会将其设置为 up 状态，该节点便成功加入了集群。

### 8.2.4 akka-cluster 集群工具

在集群运行时，及时了解所有的节点状态是非常有必要的，这通常由两种方式完成：API 及 cluster 命令行管理工具。在 API 方面，我们可以直接调用 Cluster.get(system).state() 方法得到 CurrentClusterState 对象，该对象中包含了当前 Leader 节点地址、所有节点列表、Unreachable 节点列表等信息：

```
CurrentClusterState state=Cluster.get(system).state();
System.out.println("Leader节点:"+state.getLeader());
System.out.println("所有节点列表:"+state.getMembers());
System.out.println("unreachable节点列表:"+state.getUnreachable());
```

假如要使用 cluster 命令行工具，需要先从官网下载 akka_2.11-2.4.10.zip（版本可根据实际情况选择），然后将其解压，进入 bin 目录后，你就会看到 akka-cluster 工具。使用该工具，我们不仅可以及时获取节点信息，还可以操作节点，比如 join 节点、leave 节点等。

在使用它之前，我们需要在运行时加一些 jmx 监控参数，比如：

```
java -Dcom.sun.management.jmxremote.port=7891
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false -cp
AkkaClusterDemo-1.0-allinone.jar? com.akka.book.StartUtils 2550 sys
```

在参数中，我们设置了 jmx 监听端口为 7891，假如需要在同一个 IP 上启动多个 jmx 监听，则端口不能重复。当集群启动成功后，我们就可以使用 akka-cluster 工具了。

akka-cluster 工具使用语法为：./akka-cluster hostname jmx_port command。其中 hostname 表示节点的域名或 IP，jmx_port 表示启动时指定的 jmx 监听端口，command 表示 akka-cluster 的命令，下面是一些常用命令的使用方式。

### 1. join 命令

该命令可以将节点加入集群，使用时需要指定加入的节点地址。假如要测试这个功能，我们可以定义一个没有 seed-node 节点的 Actor 系统，在启动后就可以通过如图 8-5 所示的方式将其加入集群：

```
[root@AY1212050844023222531 bin]# ./akka-cluster 127.0.0.1 7891 join akka.tcp://sys@127.0.0.1:2551
127.0.0.1 is JOINING cluster node akka.tcp://sys@127.0.0.1:2551
```

图 8-5　使用 join 命令

### 2. leave 命令

该命令可以让节点离开集群，使用时需要指定目标节点地址，如图 8-6 所示。

```
[root@AY1212050844023222531 bin]# ./akka-cluster 127.0.0.1 7891  leave akka.tcp://sys@127.0.0.1:2551
Scheduling akka.tcp://sys@127.0.0.1:2551 to LEAVE cluster
```

图 8-6　使用 leave 命令

### 3. down 命令

将节点设置为 down，使用时需要指定目标节点地址，如图 8-7 所示。

```
[root@AY1212050844023222531 bin]# ./akka-cluster 127.0.0.1 7891  down  akka.tcp://sys@127.0.0.1:2552
Marking akka.tcp://sys@127.0.0.1:2552 as DOWN
```

图 8-7　使用 down 命令

### 4. is-available 命令

检测当前节点是否可用，如图 8-8 所示。

仅供非商业用途或交流学习使用



```
[root@AY1212050844023222531 bin]# ./akka-cluster 127.0.0.1 7891 is-available
Checking if member node on 127.0.0.1 is AVAILABLE
true
```

图 8-8 使用 is-available 命令

### 5. member-status 命令

获取当前节点状态，如图 8-9 所示。

```
[root@AY1212050844023222531 bin]# ./akka-cluster 127.0.0.1 7891 member-status
Querying member status for 127.0.0.1
Up
```

图 8-9 使用 member-status 命令

### 6. members 命令

获取集群所有节点，如图 8-10 所示。

```
[root@AY1212050844023222531 bin]# ./akka-cluster 127.0.0.1 7891 members
Querying members
akka.tcp://sys@127.0.0.1:2550,akka.tcp://sys@127.0.0.1:2551,akka.tcp://sys@127.0.0.1:2552,akka.tcp://sys@127.0.0.1:2555
```

图 8-10 使用 members 命令

### 7. leader 命令

获取集群领导节点，如图 8-11 所示。

```
[root@AY1212050844023222531 bin]# ./akka-cluster 127.0.0.1 7891 leader
Checking leader status
akka.tcp://sys@127.0.0.1:2550
```

图 8-11 使用 leader 命令

### 8. cluster-status 命令

获取集群当前状态，比如集群所有成员列表及状态，当前成员地址，如图 8-12 所示。

## 8.3 集群示例——实现文章单词统计

在前面，笔者已经介绍了 Akka 集群的核心内容，但是并没有实现一个完整的功能，在本节中，笔者将实现一个单词统计的集群示例，以便让大家能更清楚地了解集群的运行过程。

### 8.3.1 示例分析

本示例将要实现的功能是：向集群发起统计单词的请求，让集群中的节点在统计好单词数后，将结果返回请求者。在这个过程中，我们需要考虑两个问题：

仅供非商业用途或交流学习使用



```
[root@AY1212050844023222531 bin]# ./akka-cluster 127.0.0.1 7891 cluster-status
Querying cluster status
{
  "self-address": "akka.tcp://sys@127.0.0.1:2555",
  "members": [
    {
      "address": "akka.tcp://sys@127.0.0.1:2550",
      "status": "Up",
      "roles": [

      ]
    },
    {
      "address": "akka.tcp://sys@127.0.0.1:2551",
      "status": "Up",
      "roles": [

      ]
    },
    {
      "address": "akka.tcp://sys@127.0.0.1:2552",
      "status": "Up",
      "roles": [

      ]
    },
    {
      "address": "akka.tcp://sys@127.0.0.1:2555",
      "status": "Up",
      "roles": [

      ]
    }
  ],
  "unreachable": [

  ]
}
```

图 8-12　使用 cluster-status 命令

1）在集群中，统计单词的节点可以有任意多个，我们将以怎样的方式和它们产生联系？

2）当集群节点中的 Actor 不可用（比如停止）时，我们将怎样感知到，并将其从待选服务列表中移除？

在这里，我们可以将该系统拆分成“前端”和“后端”两个部分来实现。当真正的客户程序访问集群时，必须先经过一个前端节点，通过该节点来管理或维护服务列表，并将任务转发到后端，当后端节点完成任务后，直接向客户程序返回统计数据（可参考图 8-13）。

如图 8-13 所示，我们可以创建三个后端节点，分别是 127.0.0.1:2550、127.0.0.1:2551、127.0.0.1:2552，当它们启动后分别向前端节点（127.0.0.1:2553）发起注册请求，表明自己已经能够开始服务了。此时前端节点记录并 watch 所有注册的后端，形成自己的“可服务列表”，当某个后端 Actor 停止时，将其移出列表。当客户程序对前端节点发起请求时，前端节点通过任务索引（JobCounter）对“可服务列表”的个数（NodeSize）取余数，以此来选择后端节点，并向其转发客户端请求。那么现在有个问题就是：怎样在集群中区分前端和后端呢？在 Akka 中，我们可以给不同的节点设置不一样的角色，该角色可以在 akka.

仅供非商业用途或交流学习使用



cluster.roles 中定义。接下来，我们来看看如何实现这个程序。

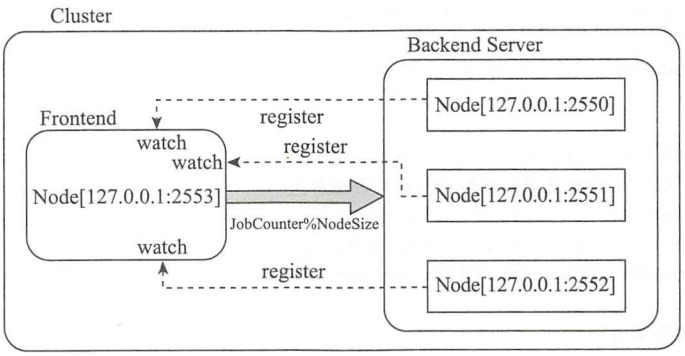



图 8-13 集群前后端架构

## 8.3.2 代码实现

首先，我们来定义前端服务。通过上面的分析可以看出，前端主要起到一个承前启后的作用，对客户程序来讲，它相当于集群的入口，所有的任务都由它转发到后端，对后端来讲，它需要维护所有可用后端，并通过合适的算法选择并委托它们进行业务处理。下面是前端 Actor 的示例代码：

```
public class WordFrontService extends UntypedActor {

    private List<ActorRef> wordCountServices = new ArrayList<ActorRef>();

    private int jobCounter = 0;

    @Override
    public void onReceive(Object msg) throws Exception {

    if (msg instanceof Article) {
        jobCounter++;
        Article art = (Article) msg;
        // 通过取余选择后端节点
        int serviceNodeIndex = jobCounter % wordCountServices.size();
        System.out.println("选择节点:"+serviceNodeIndex);
        wordCountServices.get(serviceNodeIndex).forward(art, getContext());
} else if (msg instanceof String) {
        String cmd = (String) msg;
        if (cmd.equals("serviceIsOK")) {
        // 后端Service已就绪
        ActorRef backendSender = getSender();
        System.out.println(backendSender+"可用");
        wordCountServices.add(backendSender);
        getContext().watch(backendSender);
```

仅供非商业用途或交流学习使用



```
    }else if(cmd.equals("isready")){
        if(!wordCountServices.isEmpty()){
        getSender().tell("ready",getSelf());
        }else{
        getSender().tell("notready",getSelf());
        }
    }
    } else if (msg instanceof Terminated) {
    // 当后端Service终止时，将其从后端列表中移除
    Terminated ter = (Terminated) msg;
    System.out.println("移除了"+ter.getActor());
    wordCountServices.remove(ter.getActor());
    } else {
    unhandled(msg);
    }
  }
}
/**
 * 文章
 */
class Article implements Serializable{

    private String id;
    private String content;

    public String getId() {
    return id;
    }
    public void setId(String id) {
    this.id = id;
    }
    public String getContent() {
    return content;
    }
    public void setContent(String content) {
    this.content = content;
    }
}
/**
 * 单词统计结果
 */
class CountResult implements Serializable{
    private String id;
    private int count;

    public CountResult() {
    }
    public CountResult(String id, int count) {
    this.id = id;
    this.count = count;
    }
```

仅供非商业用途或交流学习使用



```
    public String getId() {
    return id;
    }
    public void setId(String id) {
    this.id = id;
    }
    public int getCount() {
    return count;
    }
    public void setCount(int count) {
    this.count = count;
    }
}
```

为了更能体现“任务”与“处理结果”的语义，这里定义了 Article 和 CountResult 两个实体类，前者用来作为客户程序与集群之间的请求实体，后者作为集群返回给客户程序的响应实体，由于要在网络中传输它们的对象，所以需要实现 Serializable 接口，以便能进行序列化。WordFrontService 类即前端服务类，它是一个普通的 Actor。在该类中，wordCountServices 用来维护可使用的后端列表，前端在接收到后端发送过来的“ serviceIsOK”后，会将其添加到该列表并 watch 其生命周期，当后端 Actor 不可用时，前端会收到对应的 Terminated 信息，然后就可以将其移出 wordCountServices 列表。当客户程序请求 WordFrontService 时，会向其发送已经包含内容的 Article 信息，此时用 jobCounter 来记录任务的索引，然后对后端服务数取余数，以此均匀地得到一个后端节点，最后将 Article 信息转发给这个后端。这里之所以要使用 forward 而不是 tell 的原因在于，当后端处理完信息后，可以直接通过 getSender().tell 的方式将结果返回给客户程序。

后端主要做两件事情：监听集群的 MemberUp 事件，当发现有新的前端节点加入进来后，向其发送“ serviceIsOK”信息进行注册；当接收到来自前端的 Article 消息后，统计单词数并返回给客户程序，下面是示例代码：

```
public class WordCountService extends UntypedActor {

    Cluster cluster = Cluster.get(getContext().system());

    @Override
    public void preStart() {
        cluster.subscribe(getSelf(), MemberUp.class);
    }

    @Override
    public void postStop() {
        cluster.unsubscribe(getSelf());
    }

    @Override
```

仅供非商业用途或交流学习使用



```
    public void onReceive(Object msg) throws Exception {

    if (msg instanceof Article) {
        System.out.println("当前节点:"+cluster.selfAddress()+",self="+getself()+
            "正在处理……");
        Article art = (Article) msg;
        String content = art.getContent();
        int word_count = content.split("").length;
        getSender().tell(new CountResult(art.getId(), word_count),getSelf());
    }else if(msg instanceof MemberUp){
        MemberUp mu=(MemberUp)msg;
        Member m=mu.member();
        if(m.hasRole("wordFrontend")){
        getContext().actorSelection(m.address() + "/user/wordFrontService").tell
            ("serviceIsOK", getSelf());
        }
        System.out.println(m+" is up");
    }else{
        unhandled(msg);
    }
    }
}
```

为了能更加方便地测试该程序，我们可以将不同节点的端口号作为 main 函数的参数。下面代码可以用来启动前端服务：

```
public class StartFrontend {
    public static void main(String[] args) {
        String port=args[0];
        String directory=args[1];
        Config config = ConfigFactory
            .parseString("akka.remote.netty.tcp.port=" + port)
            .withFallback(
                ConfigFactory
                    .parseString("akka.cluster.roles = [wordFrontend]"))
            .withFallback(ConfigFactory.load("wordcount.conf"));
    ActorSystem system = ActorSystem.create("sys", config);
    ActorRef ref = system.actorOf(Props.create(WordFrontService.class),"wordFront-
        Service");
        String result = "";
        while (true) {
            Future<Object> fu = Patterns.ask(ref, "isready", 1000);
            try {
                result = (String) Await.result(fu,
                    Duration.create(1000, "seconds"));
            } catch (Exception ex) {
                ex.printStackTrace();
            }
            if (result.equals("ready")) {
                System.out.println("==========ready==========");
                break;
```

仅供非商业用途或交流学习使用



```
            }
        }

    List<Article> arts = new ArrayList<Article>();
        File dir = new File(directory);
        File[] files = dir.listFiles();
        try {
            for (File file : files) {
                Article art = new Article();
                art.setId(file.getName());
                StringBuffer contentBf = new StringBuffer();
                BufferedReader reader = new BufferedReader(new FileReader(file));
                String line = reader.readLine();
                while (line != null) {
                    contentBf.append(line);
                    line = reader.readLine();
                }
                reader.close();
                art.setContent(contentBf.toString());
                arts.add(art);

            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }

        Timeout timeout = new Timeout(
            Duration.create(3, TimeUnit.SECONDS));
        for (Article art : arts) {
            Patterns.ask(ref, art, timeout).onSuccess(new OnSuccess<Object>() {

                @Override
                public void onSuccess(Object res) throws Throwable {
                    CountResult cr = (CountResult) res;
                    System.out.println("文件"+cr.getId() + ",单词数:" + cr.getCount());
                }

            }, system.dispatcher());
        }
    }
}
```

在代码中，首先通过 akka.cluster.roles 将该节点设置为 wordFrontend 角色，以便让后端识别，然后等待可用的后端服务。当服务列表非空后，读取指定文件夹的文件并构造 Article 数据，然后将其发送给前端节点。当后端返回结果后，即可通过 OnSuccess 回调函数接收到。后端服务的启动过程比前端更简单，所以这里不再列出。注意，在运行测试该程序时，要先确保已经配置好集群环境，具体配置可以参考 8.2 节，这里也不再给出。图 8-14 是分别运行后端和前端后的测试结果。

仅供非商业用途或交流学习使用



```
[root@test ~]# java -cp /opt/ClusterRouter-1.0-allinone.jar com.akkajava.cr.StartFrontend 2550 /opt/data
Actor[akka.tcp://sys@127.0.0.1:2551/user/wordCountService#-1195238101]可用
Actor[akka.tcp://sys@127.0.0.1:2552/user/wordCountService#-1240265043]可用
Actor[akka.tcp://sys@127.0.0.1:2553/user/wordCountService#-26048673]可用
==========ready==========
选择节点:1
文件test1.txt,单词数:1089
选择节点:2
文件test4.txt,单词数:1089
选择节点:0
文件test6.txt,单词数:1089
选择节点:1
文件test8.txt,单词数:1089
选择节点:2
文件test2.txt,单词数:1089
选择节点:0
文件test3.txt,单词数:1089
选择节点:1
文件test5.txt,单词数:1089
选择节点:2
文件test7.txt,单词数:1089
选择节点:0
文件test9.txt,单词数:1089
选择节点:1
文件test10.txt,单词数:1089
选择节点:2
文件test11.txt,单词数:1089
选择节点:0
文件test12.txt,单词数:1089
```

前端节点 2550

```
[root@test ~]# java -cp /opt/ClusterRouter-1.0-allinone.jar com.akkajava.cr.StartBackend 2551
Member(address = akka.tcp://sys@127.0.0.1:2551, status = Up) is up
Member(address = akka.tcp://sys@127.0.0.1:2550, status = Up) is up
Member(address = akka.tcp://sys@127.0.0.1:2552, status = Up) is up
Member(address = akka.tcp://sys@127.0.0.1:2553, status = Up) is up
当前节点: akka.tcp://sys@127.0.0.1:2551,self=Actor[akka://sys/user/wordCountService#-1195238101]正在处理...
当前节点: akka.tcp://sys@127.0.0.1:2551,self=Actor[akka://sys/user/wordCountService#-1195238101]正在处理...
当前节点: akka.tcp://sys@127.0.0.1:2551,self=Actor[akka://sys/user/wordCountService#-1195238101]正在处理...
当前节点: akka.tcp://sys@127.0.0.1:2551,self=Actor[akka://sys/user/wordCountService#-1195238101]正在处理...
```

后端节点 2551

```
[root@test ~]# java -cp /opt/ClusterRouter-1.0-allinone.jar com.akkajava.cr.StartBackend 2552
Member(address = akka.tcp://sys@127.0.0.1:2550, status = Up) is up
Member(address = akka.tcp://sys@127.0.0.1:2551, status = Up) is up
Member(address = akka.tcp://sys@127.0.0.1:2552, status = Up) is up
Member(address = akka.tcp://sys@127.0.0.1:2553, status = Up) is up
当前节点: akka.tcp://sys@127.0.0.1:2552,self=Actor[akka://sys/user/wordCountService#-1240265043]正在处理...
当前节点: akka.tcp://sys@127.0.0.1:2552,self=Actor[akka://sys/user/wordCountService#-1240265043]正在处理...
当前节点: akka.tcp://sys@127.0.0.1:2552,self=Actor[akka://sys/user/wordCountService#-1240265043]正在处理...
当前节点: akka.tcp://sys@127.0.0.1:2552,self=Actor[akka://sys/user/wordCountService#-1240265043]正在处理...
```

后端节点 2552

```
[root@test ~]# java -cp /opt/ClusterRouter-1.0-allinone.jar com.akkajava.cr.StartBackend 2553
Member(address = akka.tcp://sys@127.0.0.1:2550, status = Up) is up
Member(address = akka.tcp://sys@127.0.0.1:2551, status = Up) is up
Member(address = akka.tcp://sys@127.0.0.1:2552, status = Up) is up
Member(address = akka.tcp://sys@127.0.0.1:2553, status = Up) is up
当前节点: akka.tcp://sys@127.0.0.1:2553,self=Actor[akka://sys/user/wordCountService#-26048673]正在处理...
当前节点: akka.tcp://sys@127.0.0.1:2553,self=Actor[akka://sys/user/wordCountService#-26048673]正在处理...
当前节点: akka.tcp://sys@127.0.0.1:2553,self=Actor[akka://sys/user/wordCountService#-26048673]正在处理...
当前节点: akka.tcp://sys@127.0.0.1:2553,self=Actor[akka://sys/user/wordCountService#-26048673]正在处理...
```

后端节点 2553

图 8-14　集群示例测试结果

仅供非商业用途或交流学习使用



## 8.4 使用路由

在前面的示例中，我们把集群系统分成两个部分：前端和后端，所有对集群的请求先经过前端的处理，然后根据规则转发到后端节点。实际上，在这个过程中，前端承担的主要任务就是“路由”，那么我们能否将这里的前端替换成真正的路由呢？答案是肯定的，在第5章中，笔者介绍过本地路由和远程路由的基本使用方法，而这些方法在集群中也是完全支持的。

### 8.4.1 Group 路由

Akka 集群也支持 Group 和 Pool 两种路由，前者必须建立在已经成功启动并加入集群的节点基础之上，而后者会让路由目标作为自己的子级，并且负责将其部署在远程节点。下面是一个 Group 路由的配置示例：

```
akka.actor.deployment {
/masterActor {
router=broadcast-group
routees.paths = ["/user/workerActor"]
cluster {
    enabled = on
    allow-local-routees = on
    use-role = worker
  }
 }
}
```

通过该配置，我们可以构建一个广播组的路由器，在同一个集群中，只要匹配 routees.paths 列表中的 path，都会作为路由目标。cluster.use-role 表示选用哪种角色的节点，假如没有定义，则选择所有节点。假如想要更动态地创建路由器，也可以使用如下代码：

```
List<String> routeePaths = Arrays.asList("/user/workerActor");
boolean allowLocalRoutees = true;
String useRole = "worker";
ActorRefmasterRouter = getContext().actorOf(new ClusterRouterGroup(
    new BroadcastGroup(routeePaths),
    new ClusterRouterGroupSettings(totalInstances,routeePaths, allowLocalRoutees,
        useRole))
    .props(), "masterRouter");
```

在运行 Group 类型的路由器之前，需要先保证已经启动了路由目标，即 /user/workerActor，此时你可以在多个节点上启动它们。在路由器运行的过程中，假如有新的 role=worker 的节点加入了集群，并且在该节点上启动了 path 为 /user/workerActor 的 Actor，那么它也会自动成为该路由器的路由目标。

### 8.4.2 Pool 路由

Pool 类型的路由器会将路由目标 Actor 创建在不同节点上，并管理其生命周期，当集

仅供非商业用途或交流学习使用



群中加入新的节点时，Pool 路由器会在该节点创建并启动新的路由目标 Actor，这些 Actor 会马上承担起自己的工作，而当某节点出现故障时，路由器会不断向其发送 Heartbeat 消息，并继续将任务派发到其他可用节点上的 Actor，当故障节点重新加入集群时，它也可以恢复正常工作。Pool 类型的路由器看起来如图 8-15 所示。

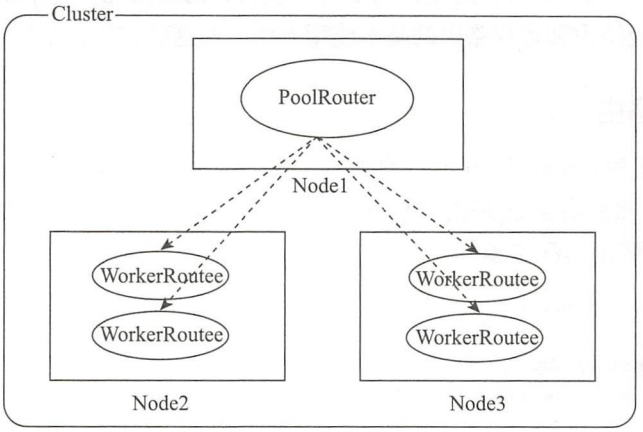



图 8-15　集群 Pool 路由器

从图 8-15 中可以看出，Pool 路由器本身具有一对多的特点，即路由器会在多个节点上创建目标 Actor。在一个集群中，路由器能创建的目标 Actor 的总数是有上限的，该上限值由 actor.deployment.default.cluster.max-total-nr-of-instances 定义，默认为 10000，你也可以自定义该配置或者在代码中重新设置。同时，路由器在每个节点上部署的最大 Actor 数也是有上限的，它的上限值由 actor.deployment.default.cluster.max-nr-of-instances-per-node 定义，默认为 1，即每个节点创建一个 Actor。假设现在将 max-total-nr-of-instances 设置为 1000，max-nr-of-instances-per-node 设置为 50，那么最多可在 20 个节点上创建目标 Actor。另外两个影响 Actor 创建的参数是 allow-local-routees 和 use-role。allow-local-routees 表示是否允许在路由器的本地节点创建目标 Actor，默认为允许（on），以图 8-15 为例，PoolRouter 将 allow-local-routees 设置为 off，所以只能在 Node2、Node3 的节点上创建 WorkerRoutee，假如将该值设置为 on，那么它也会在 Node1 上创建 WorkerRoutee。use-role 表示选择某些指定角色的节点创建 Actor，这在一定程度上可以限制路由目标的节点数。

接下来，我们用 Pool 路由改造一下“单词统计”的示例，体验一下它的便捷之处。由于路由器本身知道路由目标（即后端）的地址，所以可以把 WordFrontService 中的后端服务列表 wordCountServices 和用来做节点选取计算的变量 jobCounter 去掉，然后新建路由器 Actor，核心代码如下：

```
private ActorRef masterRouter;

    @Override
```

仅供非商业用途或交流学习使用



```
    public void preStart() throws Exception {
        int totalInstances = 100;
        int maxInstancesPerNode = 5;
        boolean allowLocalRoutees = false;
        String useRole = null;
        masterRouter = getContext().actorOf(new ClusterRouterPool(new RoundRobin
            Pool(2),
            new ClusterRouterPoolSettings(totalInstances,
            maxInstancesPerNode, allowLocalRoutees,useRole)).
props(Props.create(WordCountService.class)), "poolRouter");
    }
```

在这里，我们创建了一个支持轮询的 Pool 路由器，其中 totalInstances 表示创建的路由目标 Actor 的最大个数，maxInstancesPerNode 是每个节点创建的路由目标 Actor 的最大个数，当已有的个数大于 totalInstances 时，节点就不会再创建 Actor。然后，我们可以通过路由器来做消息转发：

```
if (msg instanceof Article) {
    Article art = (Article) msg;
    masterRouter.tell(art, getSender());
}
```

路由器将根据轮询的路由策略自动转发到目标 Actor，为了让效果更明显，我们在 WordCountService 中打印一下处理状态。分别启动前端节点（2550）、后端节点（2551、2552），最终效果如图 8-16 所示。

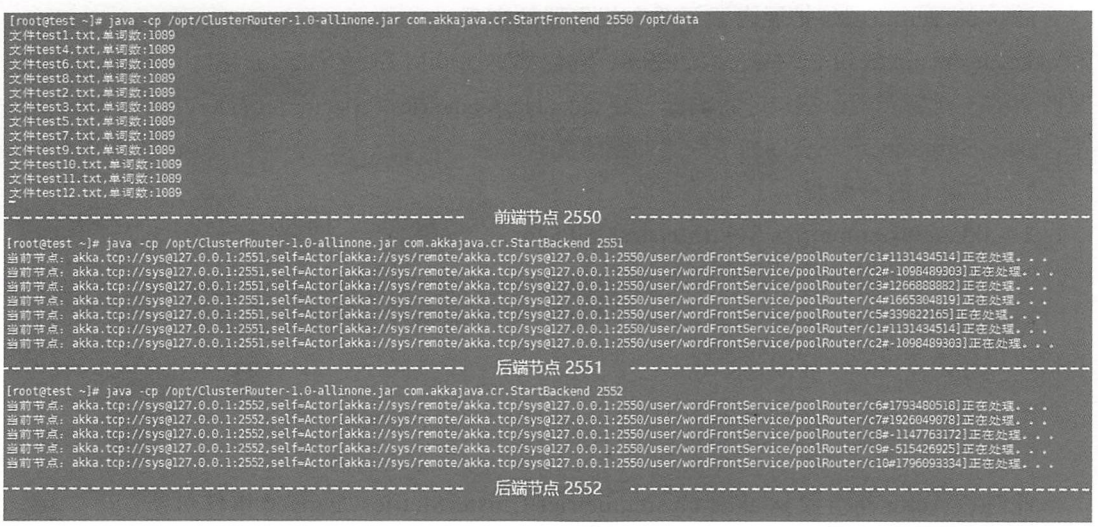



图 8-16 集群路由器测试效果

如图 8-16 所示，由于 allowLocalRoutees 设置为 false，所以在路由器节点没有创建目标 Actor。假如设置为 ture，则会按需创建。另外，由于 maxInstancesPerNode 设置为 5，

所以每个节点最多创建 5 个 Actor 实例，超出的部分会重用之前的实例。

## 8.5 集群指标

Akka 的集群环境是非常容易搭建的，在程序编写阶段，我们几乎不用考虑与集群节点任何相关的信息，基本上做到了应用程序和环境隔离。而在实际上项目中，我们需要时刻关注各个节点的指标信息，比如负载、性能或健康状况等，并以此来调优应用程序。以前面的“单词统计”为例，我们通过集群路由器将文章轮流发送到不同节点进行处理，这种方式比较常见，理论上是最为公平的路由策略，但是并没有考虑到各个节点本身的负载情况，比如我们希望将计算任务从资源紧张的节点中平衡到资源充足的节点中，那么这种路由策略肯定是办不到的，而集群的指标信息则可以作用于类似场景。

### 8.5.1 指标收集

Akka 中集群的指标收集功能是以扩展的形式存在的，它不属于核心包，所以在使用时需要加入 akka-cluster-metrics 依赖：

```
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-cluster-metrics_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
```

akka-cluster-metrics 扩展内置了两个收集器的实现 akka.cluster.metrics.SigarMetricsCollector 和 akka.cluster.metrics.JmxMetricsCollector，它们都实现了 MetricsCollector 接口，假如你想自定义收集器，就要实现该接口并将其配置在 akka.cluster.metrics.collector.provider 中。在默认情况下，akka-cluster-metrics 会以如下顺序加载收集器：

1）自定义的收集器；

2）akka.cluster.metrics.SigarMetricsCollector；

3）akka.cluster.metrics.JmxMetricsCollector。

当集群启动后，akka-cluster-metrics 组件会定时将收集到的指标信息发送到事件总线，我们可以让某个 Actor 订阅该信息：

```
ClusterMetricsExtension extension = ClusterMetricsExtension.get(system);
extension.subscribe(someActor);
```

此时该 Actor 将会收到 akka.cluster.metrics.ClusterMetricsChanged 事件对象：

```
public void onReceive(Object message) {
    if (message instanceof ClusterMetricsChanged) {
    ClusterMetricsChanged clusterMetrics = (ClusterMetricsChanged) message;
    System.out.println("======================开始========================");
```

```
        for (NodeMetrics nodeMetrics : clusterMetrics.getNodeMetrics()) {
            String info="系统信息("+nodeMetrics.address()+")\n";
            HeapMemory heapMemory =
            StandardMetrics.extractHeapMemory(nodeMetrics);
            if (heapMemory != null) {
                double heap=((double) heapMemory.used()) / (1024 * 1024);
                info=info+"Userd Heap:"+heap+" MB\n";
            }
            Cpu cpu = StandardMetrics.extractCpu(nodeMetrics);
            if (cpu != null && cpu.systemLoadAverage().isDefined()) {
                info=info+"Load:"+
                cpu.systemLoadAverage().get()+","+cpu.processors();
            }
            System.out.println(info);
        }
    System.out.println("======================结束=========================");
    }
```

在这段代码中，我们通过 ClusterMetricsChanged 对象得到了所有节点的指标信息，比如已使用的内存大小、系统平均负载（近一分钟内），处理器个数等，这些数据的采样周期由 akka.cluster.metrics.collector.sample-interval 设置，图 8-17 是该程序运行结果。

如图 8-17 所示，虽然我们最终得到了系统指标信息，但是会看到日志信息中有一些警告性的提示：当前的指标数据在某些平台上可能不正确，需要添加 sigar 包以增加准确性。这主要是因为系统在发现没有 Sigar 依赖时，转而使用了 JmxMetricsCollector 收集器，而这种收集器并不推荐使用。接下来我们看看怎么使用 Sigar 来收集指标数据。

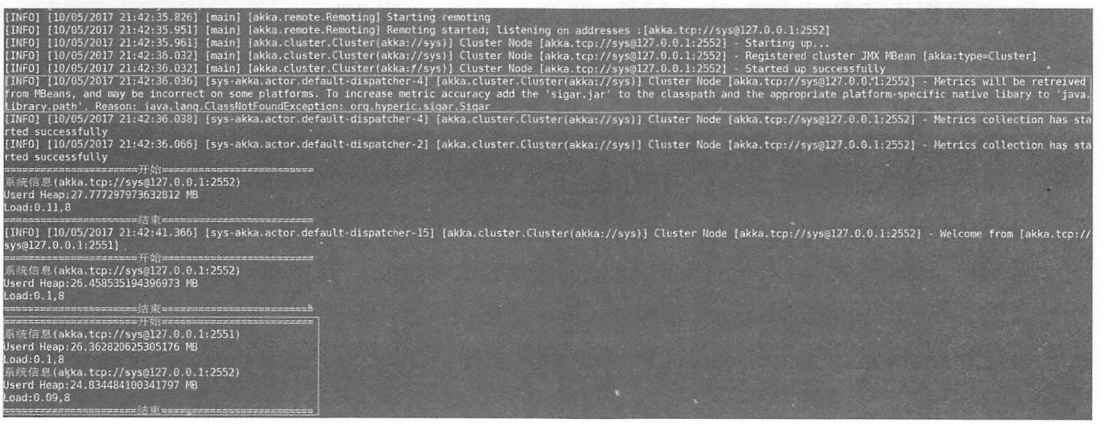



图 8-17 订阅集群指标信息

假如要直接使用 Sigar API，需要先在项目中添加它的依赖：

```
<dependency>
    <groupId>io.kamon</groupId>
    <artifactId>sigar-loader</artifactId>
```

```
    <version>1.6.6-rev002</version>
</dependency>
```

Sigar 提供了非常简单的 API 用以获取系统信息，这些信息包括：操作系统信息、内存信息、CPU 信息、文件系统信息、网络信息等。Sigar 并非一个单纯的 jar 包，它需要依赖本地库才能实现收集功能，所以在使用时不是仅仅加入 maven 依赖就行了，一个可行的方式是将 Sigar-Loader 的包下载到本地（主页 :https://github.com/kamon-io/sigar-loader），然后将其作为 Java Agent 来使用：

```
java  -javaagent:sigar-loader-1.6.6-rev002.jar -cp
AkkaMetrics  com.akkaaction.akkametrics.TestClusterMetrics
```

程序在启动时，会通过 sigar-loader 包中的 kamon.sigar.SigarAgent 将本地库解压在 native 目录下，然后加载该库。根据操作系统的不同，所解压的本地库类型也是不一样的，比如在笔者 Windows 64 位下的本地库是 sigar-amd64-winnt.dll，而 Linux 64 位下的本地库就是 libsigar-amd64-linux.so。

### 8.5.2 基于指标的负载均衡

统计集群指标的意义在于，我们希望根据系统现有情况，让资源分配得更合理、让资源利用得更充分。在 8.5 节开头我们提到一个场景，即怎样让路由器根据各个节点本身的负载情况，“智能”地将消息分派到不同的节点进行处理。现在，我们有了指标数据，就可以根据某些既定公式计算出每个节点的权重，以此作为被选择的依据，在这方面，akka-cluster-metrics 已经提供了实现，即 AdaptiveLoadBalancingPool 或 AdaptiveLoadBalancingGroup，通过前面对路由功能的了解，大家很容易通过名字可以看出它们的差别。这类路由器需要借助 MetricsSelector 接口来计算权重，该接口的实现有：

- HeapMetricsSelector：基于堆内存的权重，计算公式是：(max - used) / max，其中 max= 最大堆内存，used= 已使用堆内存；
- SystemLoadAverageMetricsSelector：基于系统平均负载（过去一分钟内）的权重，计算公式是：1-(systemLoadAverage/processors)，其中 systemLoadAverage= 系统平均负载，processors= 处理器（核）个数；
- CpuMetricsSelector：基于 CPU 利用率的权重，计算公式是：1-perc，其中 perc=CPU 利用率；
- MixMetricsSelector：综合了以上所有资源的权重。

这几个实现可以很方便地在 *.conf 文件中进行配置，示例如下：

```
akka.actor.deployment {
/clusterDemo/poolRouter??{
    router = cluster-metrics-adaptive-group
        metrics-selector = heap
```

```
    # metrics-selector = load
    # metrics-selector = cpu
    # metrics-selector = mix
}}
```

和其他类型的路由器类似，我们也可以使用硬编码的方式进行创建，以“单词统计”为例，我们可以直接将创建路由器的代码替换成如下代码：

```
masterRouter = getContext().actorOf(
new ClusterRouterPool(new
AdaptiveLoadBalancingPool(SystemLoadAverageMetricsSelector.getInstance(),0),
new ClusterRouterPoolSettings(totalInstances,
maxInstancesPerNode,allowLocalRoutees, useRole))
.props(Props.create(WordCountService.class)), "poolRouter");
```

在这段代码中，我们使用了基于系统平均负载的权重，假如要使用其他类型的权重，可以将 SystemLoadAverageMetricsSelector 换成 HeapMetricsSelector、CpuMetricsSelector 或 MixMetricsSelector 等。

## 8.6 集群单例

在本章前面的示例中，我们都是把 Actor 创建在多个节点上，每个节点也会有同类型的多个 Actor。在某些场景下，我们可能希望将某一类型的 Actor 创建为集群范围内的单例对象，比如构建一个集群任务的总路由器，或者构建一个外部系统的单点入口等，此时可以直接使用 Akka 集群的单例功能。实现一个单例 Actor 离不开 ClusterSingletonManager 的支持，它负责将创建的 Actor 部署在运行最久（oldest）的节点之上，节点运行期的比较主要依赖 Member 对象的 isOlderThan 方法。当单例 Actor 启动成功后，我们可以通过 ClusterSingletonProxy 来得到它的引用（ActorRef），并向其发送消息。当该 Actor 不可用时，ClusterSingletonProxy 会缓存发送给它的消息，直到 Actor 可用时再将消息发送过去，但是缓存的容量是有限的，假如超过限制，则会丢弃旧的消息。ClusterSingletonProxy 缓存的容量可以通过 akka.cluster.singleton-proxy.buffer-size 指定，默认为 1000，最大值为 10000。需要注意的是，在分布式环境中，我们可能还需要做大量的工作来保证消息的安全送达。

下面我们来看看到底怎样实现一个集群单例。在编写代码之前，需要先在工程中加入以下 maven 依赖：

```
<dependency>
<groupId>com.typesafe.akka</groupId>
<artifactId>akka-cluster-tools_2.11</artifactId>
<version>2.4.7</version>
</dependency>
```

然后定义 Actor，并通过 ClusterSingletonManager 创建单例对象：

```
//创建单例Actor
ClusterSingletonManagerSettings settings =
ClusterSingletonManagerSettings.create(system).withRole("worker");
    system.actorOf(ClusterSingletonManager.props(
    Props.create(SingleActorDemo.class), PoisonPill.getInstance(),settings),
"singleActor");
```

ClusterSingletonManagerSettings 的 withRole 方法用于设置目标节点的角色，假如希望覆盖所有节点，则不必调用该方法。为了更好地观察测试效果，我们可以在 SingleActorDemo 中打印 MemberUp 信息，当多个节点启动时，你就会看到单例对象只在某一个节点上启动了。为了得到集群中的单例对象，我们需要先创建代理配置对象（ClusterSingletonProxySettings），通过它可以设置角色、消息缓冲区大小等参数，然后使用 ClusterSingletonProxy 构造 Props 工厂并委托 actorOf 创建 Actor 代理对象：

```
//通过代理得到单例Actor引用
    ClusterSingletonProxySettings proxySettings =
    ClusterSingletonProxySettings.create(system).withRole("worker");
    ActorRef
client=system.actorOf(ClusterSingletonProxy.props("/user/singleActor", proxySettings),
    "singleProxy");
```

单例 Actor 虽然简单，但是切记不要滥用，因为它可能会产生单点故障，以及单点带来的性能瓶颈。

## 8.7 集群客户端

### 8.7.1 集群客户端概述

集群内的 Actor 可以通过很多方式进行通信，比较典型的方式就是使用路由，但是这种方式仅仅只对同一个集群内的 Actor 起作用，如果你想在外部（比如另一个集群）访问集群，则需要使用 ClusterClient。和直接访问远程 Actor 的方式不同，使用 ClusterClient 时需要先通过集群连接点（contact point）来建立与 ClusterReceptionist 的连接，然后使用后者与集群内部注册过的 Actor 进行通信，图 8-18 展示了这一过程。

如图 8-18 所示，集群客户端的使用需要服务端的支持。当服务端启动后，它需要通过 ClusterClientReceptionist 在每个节点创建一个 ClusterReceptionist 类型的 Actor，这个 Actor 主要负责将客户端的消息转发给集群中已注册的 Actor，一般来讲，我们把 ClusterReceptionist Actor 称为 receptionist，即接待员。在 receptionist 内部引用了 DistributedPubSubMediator 类型的订阅器，当 ClusterClientReceptionist 注册需要被外部访问的 Actor 时，会向该订阅器发送 DistributedPubSubMediator.Put(actorRef) 消息，此时会把 Actor 加入订阅器的 registry 列表并 watch 其生命周期，当该 Actor 终止时，订阅器会自动将其从 registry 列表中移出。

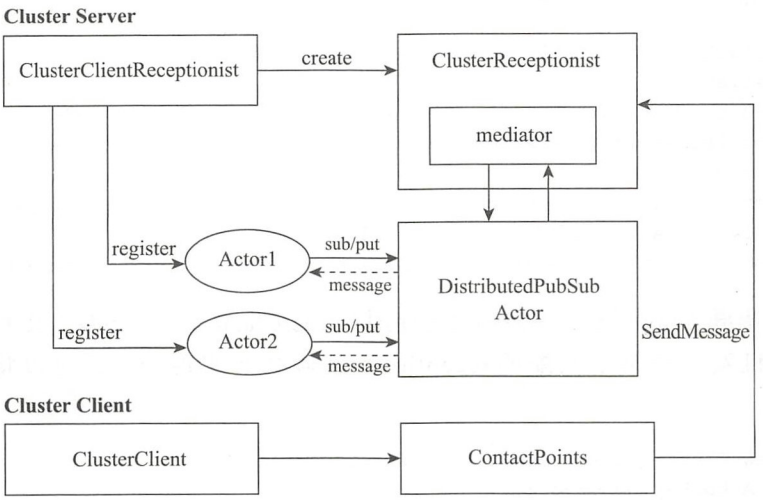



图 8-18 集群客户端使用过程

客户端在访问集群时并不需要知道每个 Actor 的详细地址，但是它必须知道集群节点上的 receptionist 地址，因为它正是通过 receptionist 向集群中的 Actor 发送消息的。在使用 ClusterClient 创建客户端 Actor 时，我们需要明确指定 contact-points，它包含了一个或多个集群节点的地址，以便能找到集群中的 receptionist。当客户端 Actor 向集群发送消息时，receptionist 会通过 DistributedPubSubMediator 将信息转发给注册过的 Actor。

## 8.7.2 使用 ClusterClient

在使用 ClusterClient 之前，需要先确保项目中已加入 akka-cluster-tools_2.11 依赖，具体 Maven 配置可参考前面章节，这里不再赘述。

现在，我们先看看服务端需要做哪些工作。在前面，我们已经了解到：要让某个 Actor 能被外部访问，需要先使用 ClusterClientReceptionist 创建 receptionist（即 ClusterReceptionist 类型的 Actor）。ClusterClientReceptionist 本身是一个扩展实现，我们可以直接配置在 *.conf 文件里：

```
akka.extensions = ["akka.cluster.client.ClusterClientReceptionist"]
```

当 ActorSystem 在多个节点启动时，它就会自动地在每个节点创建 receptionist，实际上，receptionist 就是一个普通的 Actor。接下来，我们可以调用 ClusterClientReceptionist 的 registerService 方法对 Actor 进行注册：

```
ActorRef userActor=system.actorOf(Props.create(UserActor.class),"userActor");
ClusterClientReceptionist.get(system).registerService(userActor);
```

客户端的工作也比较简单，首先我们需要定义一系列 contact-points，然后创建用于和

服务端 receptionist 通信的代理 Actor：

```
Set<ActorPath> initContacts = new HashSet<ActorPath>(
Arrays.asList(
ActorPaths.fromString("akka.tcp://sys@127.0.0.1:2550/system/receptionist"),
ActorPaths.fromString("akka.tcp://sys@127.0.0.1:2551/system/receptionist")));
ActorRef c =
system.actorOf(ClusterClient.props(ClusterClientSettings.create(system)
.withInitialContacts(initContacts)), "client");
c.tell(new ClusterClient.Send("/user/userActor", "hello", true),ActorRef.noSender());
```

注意，向集群 Actor 发送的消息需要先使用 ClusterClient 做一层包装，其中 ClusterClient.Send 表示随机发送给某个匹配的 receptionist，假如要实现群发，可以将消息包装成 SendToAll：

```
c.tell(new ClusterClient.SendToAll("/user/userActor",
"hello"),ActorRef.noSender());
```

假如客户端使用了单独的 *.conf 文件，那么你需要将 akka.actor.provider 配置成 akka.remote.RemoteActorRefProvider 或者 akka.cluster.ClusterActorRefProvider，以便能进行远程访问。

### 8.7.3 事件监听

集群的服务端和客户端都能够通过订阅事件来得到对方的连接信息。对于服务端来讲，它可以通过向 receptionist 发送 SubscribeClusterClients.getInstance() 消息来订阅客户端的连接信息，包括客户端列表、新增客户端连接、丢失某客户端连接（unreachable）等；对于客户端来讲，它可以通过向客户端 Actor 发送 SubscribeContactPoints.getInstance() 消息来订阅服务端 receptionists 的信息，包括 contact-point 列表、新增 contact-point、删除 contact-point 等。下面是服务端和客户端获取事件信息的部分核心代码：

```
/**
 * 监听ClusterClient信息
 * @author dyf
 */
class ServerListener extends UntypedActor {
    @Override
    public void onReceive(Object message) {
        if (message instanceof ClusterClients) {
            System.out.println("ClusterClients:"+message);
        } else if (message instanceof ClusterClientUp) {
            //可在此添加ClusterClient
            System.out.println("ClusterClientUp:"+message);
        } else if (message instanceof ClusterClientUnreachable) {
            //可在此删除某个ClusterClient?
            System.out.println("ClusterClientUnreachable:"+message);
        }
    }
```

```
}

/**
 * 监听ClusterClientReceptionist消息
 * @author dyf
 *
 */
class ClientListener extends UntypedActor {

    @Override
    public void onReceive(Object message) {
        if (message instanceof ContactPoints) {
            System.out.println("ContactPoints: "+message);
        } else if (message instanceof ContactPointAdded) {
            //可在此添加contact-point
            System.out.println("ContactPointAdded: "+message);
        } else if (message instanceof ContactPointRemoved) {
            //可在此删除contact-point
            System.out.println("ContactPointRemoved: "+message);
        }
    }
}
```

## 8.8 集群分片

### 8.8.1 分片概念

分片的目的在于将 Actor 均匀地分布在集群中的各个节点上。一个集群往往存在多个分片（shard），而每个分片又包含多个 Actor，这些 Actor 被称为 entity，它们关联到一个唯一标识符，即 entityId。分片中的 Actor 地址透明，所以在引用它们时不必知道其具体地址，只需要知道 entityId 即可。entityId 是字符串类型的，其格式并没有特别要求，你既可以简单将其设置为 UUID，也可以设置为某个业务 id，比如用户 id 或者 session id 等。

不是所有 Actor 都能自动地进行分片，你需要先将目标 Actor 的类型注册在 ClusterSharding 中，然后在各个节点中启动它们。当访问这类 Actor 时，需要先使用 ClusterSharding 得到 ShardRegion 对象，然后通过 ShardRegion 对象和目标 Actor 进行通信。由于目标 Actor 都处于某个分片，所以访问它们前，必须先知道分片所在的位置，在底层实现中，这个工作交给 ShardCoordinator 来做，它是一个单例 Actor，所有节点都可以访问它并得到分片位置的一致性视图。图 8-19 展示了集群分片的基本结构。

当 Actor 被注册到分片中后，它不会马上创建或启动。在通过 ShardRegion 向这些 Actor 发送消息时，系统会根据你设定的规则来提取 shardId 和 entityId。得到 shardId 后，系统就能找到对应的分片；得到 entityId 后，系统会根据它来查找或创建对应的 Actor，并将其均匀地分布在各个节点中。

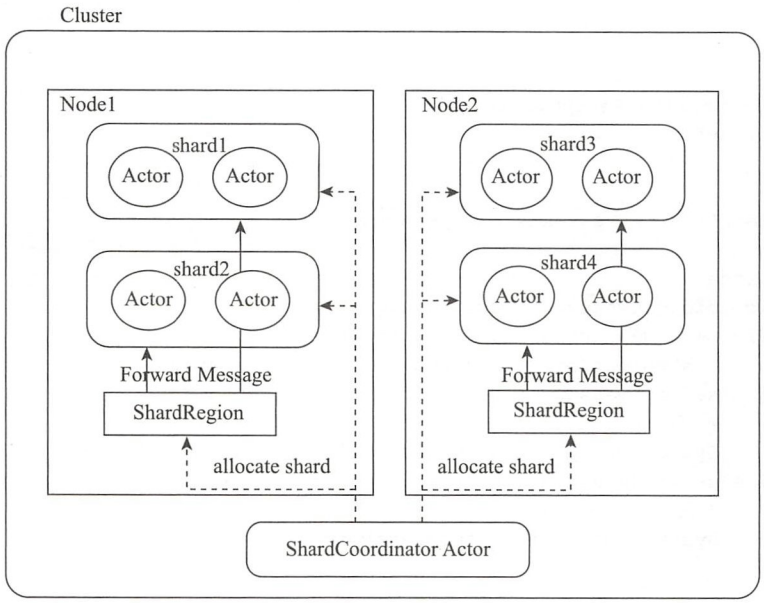



图 8-19　集群分片基本结构

一般来说，分片中的 Actor 都是有状态的，为了保证集群下的高可用，通常会将其设计成持久化 Actor，以便在集群节点崩溃后自动恢复。负责管理分片位置的 ShardCoordinator 是一个集群单例 Actor，当它出现故障时，意味着所有分片的位置都会丢失。不过我们不用担心，在底层创建 ShardCoordinator 时，会使用其子类 PersistentShardCoordinator 来实现，而 PersistentShardCoordinator 就是一个持久化的 Actor，当节点出现故障时，集群会创建一个新的 ShardCoordinator 来接管工作，并且根据之前持久化的数据来恢复状态。

由于持久化是集群分片中一个不可或缺的要素，所以在本节中，笔者将首先介绍持久化 Actor 的基础用法，然后再用一个“购物”示例来讲解分片的使用。

## 8.8.2　持久化 Actor

Akka 的持久化功能可以让 Actor 保存它们的内部状态，以便在程序出现故障或者 JVM 崩溃时进行自我恢复。要注意的是，Akka 持久化并不会直接保存 Actor 的当前状态，而是会保存所有变化，当需要恢复时，Actor 会通过回放这些变化来重建自己的状态，这个过程遵循 EventSourcing 的设计理念。

EventSourcing 是一种以事件为单位的架构模型，具体来讲就是：当业务对象在处理某个逻辑之后，会把这个处理过程相关的数据抽象并保存起来，这些数据包括时间、状态、动作等，而这些数据组成的对象我们可称之为事件。也就是说，事件对象描述的就是过去发生的每个行为以及这个行为产生的数据。当事件对象被保存成功后就不能再修改，只能继续追加。

一个复杂的业务逻辑可能会包含很多个事件对象，在必要的情况下，我们可以通过这些事件对象来回溯整个业务过程，但前提是，这些事件已经处于持久化的状态，而持久层既可以是文件，也可以是数据库。

在真正实现 EventSourcing 之前，我们需要根据业务来定义好事件命令和状态。

### 1. 定义持久化 Actor

在 Akka 中，EventSourcing 已在持久化 Actor 中得到支持。由于 Akka 持久化是独立的模块，所以在使用时先确保已加入 maven 依赖：

```
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-persistence_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
```

该模块提供的 UntypedPersistentActor 即持久化 Actor，它拥有 onReceiveCommand 和 onReceiveRecover 两个关键方法，前者用于接收外部消息（和普通 Actor 的 onReceive 方法类似），一般情况下，我们会在这个方法里面调用 persist 方法来保存事件对象；后者主要用于执行数据恢复逻辑，它会在 Actor 重启后自动调用。除了保存单个事件外，我们还可以调用 saveSnapshot 方法来保存当前状态下的快照，使用快照的好处是可以大幅度缩短恢复时间。下面看一个完整的示例：

```
class UserService extends UntypedPersistentActor {
    /**
     * 状态列表
     */
    private List<Action> states = new ArrayList<Action>();
    @Override
    public String persistenceId() {
    return "userservice-1";
    }
    @Override
    public void onReceiveCommand(Object message) throws Exception {
    if (message instanceof Action) {
    Action action = (Action) message;
    if (action.getCmd().equals("save")) {

    /**
    * 持久化事件，异步执行
    */
    persist(action, new Procedure<Action>() {
    @Override
    public void apply(Action act) throws Exception {
    // 通常会在这里更新Actor的状态，或者发布事件通知订阅者
    states.add(act);
    }
    });
```



```
        } else if (action.getCmd().equals("saveAll")) {
        /**
         * 保存快照
         */
        saveSnapshot(states);
        } else if (action.getCmd().equals("get")) {
        System.out.println("state:" + states);
        }
        } else if (message instanceof SaveSnapshotSuccess) {
        SaveSnapshotSuccess saveSnapSucc = (SaveSnapshotSuccess) message;
        System.out.println("save snap success:" + saveSnapSucc.metadata());
        } else {
        System.out.println("other message " + message);
        }
        }
        @Override
        public void onReceiveRecover(Object msg) throws Exception {
            if (msg instanceof Action) {
                Action evt = (Action) msg;
                states.add(evt);
        } else if (msg instanceof SnapshotOffer) {
            SnapshotOffer snapOffer = (SnapshotOffer) msg;
            states = (List<Action>) snapOffer.snapshot();
            System.out.println("recover: " + states);
        }else if(msg instanceof RecoveryCompleted){
            System.out.println("replay has been finished");
        }
            System.out.println("onreceiveRecover:" + msg + "," + msg.getClass());
        }
    }
    class Action implements Serializable{
        private String cmd;
        private String data;
        public Action(String cmd, String data) {
        this.cmd = cmd;
        this.data = data;
        }
    /*getter setter省略*/
      @Override
      public String toString() {
      return this.cmd + "--->" + this.data;
      }
    }
```

在这个例子中，我们首先定义了一个用于处理用户信息的持久化 Actor。其中 persistenceId 方法用于定义 Actor 的持久化 id，它是唯一的。在 onReceiveCommand 方法中，首先匹配消息类型，假如是 save 操作，则调用 persist 方法保存事件对象，一旦保存成功，就会调用 Procedure 方法，此时我们可以在这里更新当前 Actor 的状态，或者发布事件通知订阅者。在持久化处理的过程中，Actor 不会接收新的消息。要注意的是，持久的消息必须配置序列化，或



者直接实现序列化接口，否则会出现保存或恢复失败。当向 Actor 发送 saveAll 指令时，我们调用 saveSnapshot 方法保存了当前快照，假如成功会收到 SaveSnapshotSuccess 消息，否则会收到 SaveSnapshotFailure 消息。图 8-20 展示了向 UserService 发送两次 save/get 命令的运行效果。

```
replay has been finished
onreceiveRecover:RecoveryCompleted,class akka.persistence.RecoveryCompleted$
[WARN] [12/04/2017 22:37:33.095] [sys-akka.persistence.dispatchers.default-plugin
e another serializer or disable this warning using the setting 'akka.actor.warn-a
state:[save--->001]
state:[save--->001, save--->002]
```

图 8-20 向 UserService 发送 save/get 命令

持久化 Actor 在启动（或重启）时会自动回放已经保存过的事件，此时我们可以在 onReceiveRecover 方法中收到这些事件并自定义恢复策略，当所有事件都被回放后，我们会收到 RecoveryCompleted 消息，表示恢复完成，图 8-21 是重启 UserService 后的恢复效果。

```
onreceiveRecover:save--->001,class org.pers.Action
onreceiveRecover:save--->002,class org.pers.Action
replay has been finished
onreceiveRecover:RecoveryCompleted,class akka.persistence.RecoveryCompleted$
state:[save--->001, save--->002]
```

图 8-21 UserService 重启并获取最新状态

如果仅仅只需要重放前 n 个事件，则可以重写 recovery 方法并调用 Recovery. create(n) 即可。假如之前保存过快照，在恢复时也会收到 SnapshotOffer 消息，调用其 snapshot 方法即可返回快照对象，通过它可以直接更新当前 Actor 的状态。当恢复失败时，系统会自动调用 onRecoveryFailure 方法并停止该 Actor，这里之所以不直接恢复运行，是因为在出现这种错误时，我们无法得知持久化是否已按照既定的方式执行了。

### 2. 插件配置

讲到这里，大家心里可能会有一个疑问：既然持久化 Actor 能够在重启后回放之前保存的事件，说明这些事件肯定不是存储在内存中，那么会存储在哪儿呢？实际上，Akka 已经提供了用于保存事件日志和快照的插件，我们只需要在 *.conf 文件中配置这些插件即可，比如：

```
akka.persistence.journal.plugin = "akka.persistence.journal.leveldb"
akka.persistence.snapshot-store.plugin =
"akka.persistence.snapshot-store.local"
akka.persistence.journal.leveldb.dir = "target/example/journal"
akka.persistence.snapshot-store.local.dir = "target/example/snapshots"
```

akka.persistence.journal.plugin 指定了事件日志的插件，这里采用了 LevelDB 作为底层存储，当调用 persist 方法后，事件将被存储在 akka.persistence.journal.leveldb.dir 指定的路径下。

akka.persistence.snapshot-store.plugin 指定了快照存储的插件，当调用 saveSnapshot 方法后，快照将被存储在 akka.persistence.snapshot-store.local.dir 指定的路径下。



另外，由于 LevelDB 并没有集成在 Akka-Persistence 模块中，所以在使用时还得在 Maven 中加入相关依赖：

```
<dependency>
    <groupId>org.iq80.leveldb</groupId>
    <artifactId>leveldb</artifactId>
    <version>0.7</version>
</dependency>
<dependency>
    <groupId>org.fusesource.leveldbjni</groupId>
    <artifactId>leveldbjni-all</artifactId>
    <version>1.8</version>
</dependency>
```

假如大家觉得它自带的插件并不符合项目需要，可以考虑去 http://akka.io/community/ 获取其他存储插件。当然，你也可以自定义插件，比如对于事件日志，你可以继承 AsyncWriteJournal 并实现其相关方法，然后将其配置在 *.conf 文件里：

```
akka.persistence.journal.plugin = "custom-journal"
custom-journal{
    class = "com.book.CustomJournal"
    /*其他配置略*/
}
```

假如要自定义快照插件，则需要继承 SnapshotStore，它的配置方式可参照上面的做法，大家可以自行尝试，这里不再赘述。

### 8.8.3 分片示例

在实际项目中，我们经常会遇到需要保存用户状态的场景，比如用户的登录状态，或者用户的购物车列表等，在 Web 系统里面，这些都可以很轻易地使用 session 或 cookie 机制来实现。以购物车为例，当用户第一次访问站点时，系统会生成一个 sessionId 作为唯一标识，只要用户在随后的访问中附带上这个标识，系统就能匹配到该用户之前的 session 信息。当然，一般的 Web 服务器都已经实现了 session 功能，我们不必重复造这个轮子。刚才这个 session 场景实际上可以用分片 Actor 来模拟，比如说，我们可以让 sessionId 作为 Actor 的 entityId，当第一次访问这个 Actor 时，它可能还并不存在，此时分片系统会根据 entityId 来创建并启动一个 Actor，当用户再次访问时，可以在消息里附带上 entityId，那么它就能命中同一个 Actor。假如需要在每次操作后都保存当前状态，可以使用持久化 Actor，此时你无须担心由于存储而产生的资源消耗，因为每个 Actor 都会均匀地分布在不同节点（机器）的区域中。

接下来，我们就用集群分片实现一个“购物车”功能，图 8-22 展示了该购物车的基本架构。

该购物车主要有两个功能：购买和查询。当用户发起购买时，会向系统发送“buy”指令并携带商品信息，系统接收到请求后会委托到某个持久化 Actor 进行存储；当用户发起查询时，会向系统发送“query”指令，此时系统会找到对应的 Actor 并获取之前存储的购



物列表。所有对购物车的请求都被封装成一个 Cmd 对象，该对象主要包含两个数据：用户 ID(userId) 和动作 (action)，也可能还有其他请求数据。当用户发起请求时，ShardRegion 会根据自定义的规则对 Actor 进行分片处理，这个规则由 MessageExtractor 定义，它拥有两个关键方法：entityId 和 shardId。entityId 方法主要用于从消息中提取 entityId 值，对于第一次请求，系统会在合适的节点创建远程 Actor，该 Actor 的名字即为 entityId，此时你不必关心它的具体位置，在进行后续请求时，系统会根据提取的 entityId 重新查找到正确的 Actor，在我们这个例子中，笔者将 userId 作为 entityId。shardId 方法主要用于提取消息中的 shardId（分片标识）值，它是创建和查找分片的依据，为了简单起见，我们通过对分片个数取模的方式来生成 shardId。

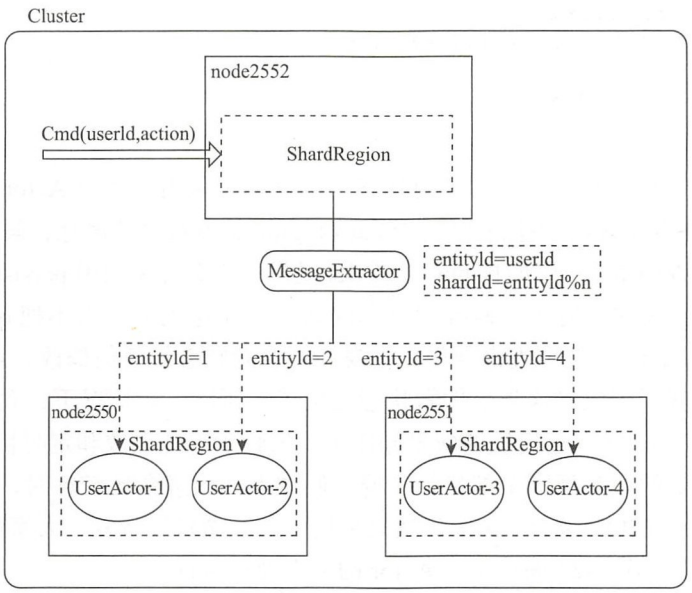



图 8-22 “购物车”分片基本架构

在理解了这个示例的基本运行过程之后，我们来看看怎样编码实现它。由于集群分片不属于 Akka 核心模块，所以在编写代码之前先要确保项目中已加入以下 maven 依赖：

```
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-cluster-sharding_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
```

首先，我们来定义消息（指令）的格式。由于分片操作依赖于消息中提取的 shardId 和 entityId，所以在每次请求时，必须一直持有能生成这两个 id 的数据，在本例中，笔者直接将 userId 作为 entityId。用户可能会对购物车发起多个不同类型的请求，为了让系统能识别每种请求并做相应处理，我们可以定义一个 action 字段来描述请求的动作，它的可选值有：

buy（购买）、query（查询）、del（删除）。下面是这个消息类的基础代码：

```
/**
 * 指令
 * query:查询购物车
 * buy:购买item
 * del:从购物车删除item
 * @author dyf
 */
public class Cmd implements Serializable {
    public final static String BUY="buy";
    public final static String QUERY="query";
    public final static String DEL="del";
    private String action;
    private int userid;
    //商品对象，拥有id、name、price属性，具体代码略
    private Item item;
    //getter setter省略...
}
```

由于后端需要长时间维护购物车的状态，所以我们采用持久化 Actor 来处理用户的请求。这个 Actor 主要实现的功能是：接收 ShardRegion 转发过来的消息，假如是 query 操作，则向 sender 返回查询结果；假如是 buy 或者 del 操作，则会首先调用 persist 方法将这个（事件）命令持久化，成功之后再去修改购物车的状态。那这里为什么不把 query 也做持久化呢？原因就在于 query 只是个查询操作，本身并不会对状态做任何修改，Actor 在恢复阶段做事件重放时也就不用依赖这类事件做状态变更了，所以一般情况下，我们只会持久化能改变状态的事件。Actor 在重启后的恢复操作主要依赖于事件重放和定时快照，前者需要让所有事件按照既定的业务规则全部执行一遍，假如事件太多会比较耗时，而后者则会大幅度缩短恢复时间，在我们这个示例中仍然只采用了前者做状态恢复，大家可以参考 8.8.2 节自行添加快照功能。图 8-23 展示了该 Actor 的基本处理流程。

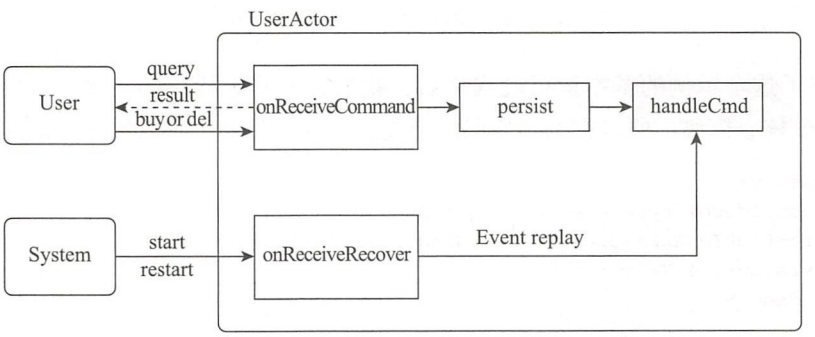



图 8-23　UserActor 消息处理及恢复流程

如图 8-23 所示，由于在恢复阶段也需要执行相同的业务流程，所以这里将该流程封装在 handleCmd 方法里，以便让其重用。下面是 UserActor 的实现代码：

```
public class UserActor extends UntypedPersistentActor {
    /**
     * 购物车列表
     */
    private Map<Integer, Item> cart = new HashMap<Integer, Item>();
    Cluster cluster = Cluster.get(getContext().system());
    public UserActor() {
    context().setReceiveTimeout(Duration.create(60, TimeUnit.SECONDS));
    }
    @Override
    public void preStart() throws Exception {
    super.preStart();
    System.out
    .println(cluster.selfAddress() + "." + getSelf() + " start");
    }
    @Override
    public String persistenceId() {
    return "user_" + getSelf().path().name();
    }
  @Override
    public void onReceiveCommand(Object message) throws Exception {
    if (message instanceof Cmd) {
    System.out.println("cmd: " + message);
    Cmd cmd = (Cmd) message;
    if (cmd.getAction().equals(Cmd.QUERY)) {
    getSender().tell(cmd.getUserid() + " cart:" + cart,getSelf());
    }else{
    persist(cmd, new Procedure<Cmd>() {
    public void apply(Cmd cmd) {
    handleCmd(cmd);
    }});
}}else if (message.equals(ReceiveTimeout.getInstance())){
    getContext().parent().tell(
    new ShardRegion.Passivate(PoisonPill.getInstance()),
    getSelf());
    }else{
    System.out.println("other msg: "+message);
}}
    private void handleCmd(Cmd cmd) {
    if (cmd.getAction().equals(Cmd.BUY)) {
    Item item = cmd.getItem();
    cart.put(item.getId(), item);
    } else if (cmd.getAction().equals(Cmd.DEL)) {
    Item item = cmd.getItem();
    cart.remove(item.getId());
    }else{
    System.out.println("other handlerCmd: "+cmd);
    }}
    @Override
    public void onReceiveRecover(Object message) throws Exception {
    if (message instanceof Cmd) {
```



仅供非商业用途或交流学习使用

```
        Cmd cmd = (Cmd) message;
        handleCmd(cmd);
        }}}
```

每个持久化 Actor 都需要一个唯一的 persistenceId，在 UserActor 中，我们指定了 getSelf().path().name() 作为它的 persistenceId，这个 name 实际上也是它的 entityId。UserActor 使用了一个 Map 结构来存储购物列表，当 Actor 接收到 query 消息时将直接返回此列表给 sender。当 Actor 接收到 buy 或 del 消息时，由于涉及了状态修改，所以先是调用 persist 方法持久化相关事件，成功之后再调用 handleCmd 方法修改购物车的状态，persist 方法是异步执行的，但是它能保证在当前处理完成之前不会处理下一个消息。由于在 Actor 创建阶段我们设置了 ReceiveTimeout，所以一旦它在设定时间内不能继续收到外部的 Cmd，那么它就会收到 ReceiveTimeout.getInstance() 消息，此时我们向它的父级 shard 发送 ShardRegion.Passivate 消息，使 Actor 被钝化（Passivate）处理，紧接着 Actor 就会被终止，并且新进来的消息将会被缓冲到分片中以便能转发给新的 Actor。

如 8.8 节所说，Actor 只有被注册在 ClusterSharding 中才能进行分片。注册由 ClusterSharding.start 方法实现，该方法需要 typeName、Props、MessageExtractor 等参数，其中 typeName 是自定义的 shardRegion 名称，它是后面获取 shardRegion 的依据；Props 是用于创建 Actor 的配置对象；MessageExtractor 主要用于从消息中提取 entityId 和 shardId。下面是注册分片 Actor 的核心代码：

```
ShardRegion.MessageExtractor messageExtractor = new ShardRegion.
MessageExtractor() {
    @Override
    public String entityId(Object message) {
        String entityId = null;
            if (message instanceof Cmd) {
                Cmd cmd = (Cmd) message;
                entityId = cmd.getUserid() + "";
            }
            return entityId;
    }
    @Override
    public Object entityMessage(Object message) {
        return message;
    }
    @Override
    public String shardId(Object message) {
        String shardId = null;
        int numberOfShards = 10;
        if (message instanceof Cmd) {
            Cmd cmd = (Cmd) message;
            shardId = (cmd.getUserid() % numberOfShards) + "";
        }
        return shardId;
    }
};
```



仅供非商业用途或交流学习使用

```
ClusterShardingSettings settings = ClusterShardingSettings.create(system);
ClusterSharding.get(system).start("userActor",Props.create(UserActor.class),
settings, messageExtractor);
```

在 MessageExtractor 里面，我们直接将 Cmd 消息中的 userId 作为 entityId，并且通过让 userId 对 shard 个数取余来得到 shardId，这种方式基本满足了 Actor 的均匀分布。MessageExtractor 的 entityMessage 方法主要用于重新提取或包装发往 Actor 的消息，一般情况下，我们会直接转发原消息。假如想要更简单的实现 MessageExtractor，可以考虑直接使用 HashCodeMessageExtractor，它需要我们自己实现 entityId 方法，然后在内部默认通过 entityId.hashCode%maxNumberOfShards 计算出 shardId，其中 maxNumberOfShards 表示最大 shard 数，该值在对象创建时传入。

当调用 start 方法时，ClusterSharding 会创建一个 ClusterShardingGuardian 类型的 Actor 并向其请求（通过发送 Start 消息）ShardRegion Actor，然后将得到的 ShardRegion 存入一个 Map 结构，以便在后续重新 get 出来。ClusterShardingGuardian Actor 的名字由 akka.cluster.sharding.guardian-name 指定，默认为 sharding，所以它的 path 为 /system/sharding。ShardRegion 和 ShardCoordinator 都是这个 Actor 的子级。

ClusterSharding 是一个扩展实现，它需要在每个节点启动。要注意的是，在调用 start 方法后系统不会立即创建目标 Actor，这些 Actor 都是按需创建的。为了能访问目标 Actor，我们需要先通过 ClusterSharding 得到一个 ShardRegion 对象，下面是部分测试代码：

```
ActorRef userRegion= ClusterSharding.get(system).shardRegion("userActor");
if("buy".equals(command)){
        //购买
        Cmd cmd0 = new Cmd(Cmd.BUY, 0);
        cmd0.setItem(new Item(0, "Java编程思想", 80));
        //actorClient是普通Actor，用于接收对方返回的数据，具体代码略...
        userRegion.tell(cmd0, actorClient);

        Cmd cmd1 = new Cmd(Cmd.BUY, 1);
        cmd1.setItem(new Item(1, "C++编程思想", 90));
        userRegion.tell(cmd1, actorClient);

        Cmd cmd2 = new Cmd(Cmd.BUY, 2);
        cmd2.setItem(new Item(2, "Akka实战", 60));
        userRegion.tell(cmd2, actorClient);
    }else if("query".equals(command)){
        //查询
        userRegion.tell(new Cmd(Cmd.QUERY, 0), actorClient);
        userRegion.tell(new Cmd(Cmd.QUERY, 1), actorClient);
        userRegion.tell(new Cmd(Cmd.QUERY, 2), actorClient);
    }
```

其中 shardRegion 方法中传入的“userActor”即为之前注册过的 typeName。当通过 ShardRegion 发送消息时，它会提取目标 Actor 的分片位置，假如此时在分片中找不到这个 Actor，则会创建一个，后面就可以继续按照 entityId 规则命中到正确的 Actor 了，在这整



仅供非商业用途或交流学习使用

个过程你都不需要知道对方的具体地址。

为了能更好地观察持久化事件日志的存取过程，可以配置一款持久化插件，比如 akka-persistence-jdbc，它支持基于 JDBC 的事件存储，它的主页是 https://index.scala-lang.org/dnvriend/akka-persistence-jdbc。

在配置 maven 依赖时，注意选择合适的版本，笔者这里选用 2.8.0，如下：

```
<dependency>
    <groupId>com.github.dnvriend</groupId>
    <artifactId>akka-persistence-jdbc_2.11</artifactId>
    <version>2.8.0</version>
</dependency>
```

另外，在 *.conf 中需要配置插件相关的信息，包括数据库地址、账号等，这个可以参考：

https://github.com/dnvriend/akka-persistence-jdbc/blob/master/src/test/resources/mysql-application.conf

然后，在 mysql 库中创建 journal 和 snapshot 两个表，它们分别用于存储日志和快照，建表语句可以参考：https://github.com/dnvriend/akka-persistence-jdbc/blob/master/src/test/resources/schema/mysql/mysql-schema.sql。

图 8-24 和图 8-25 展示了启动及访问分片 Actor 的测试效果。

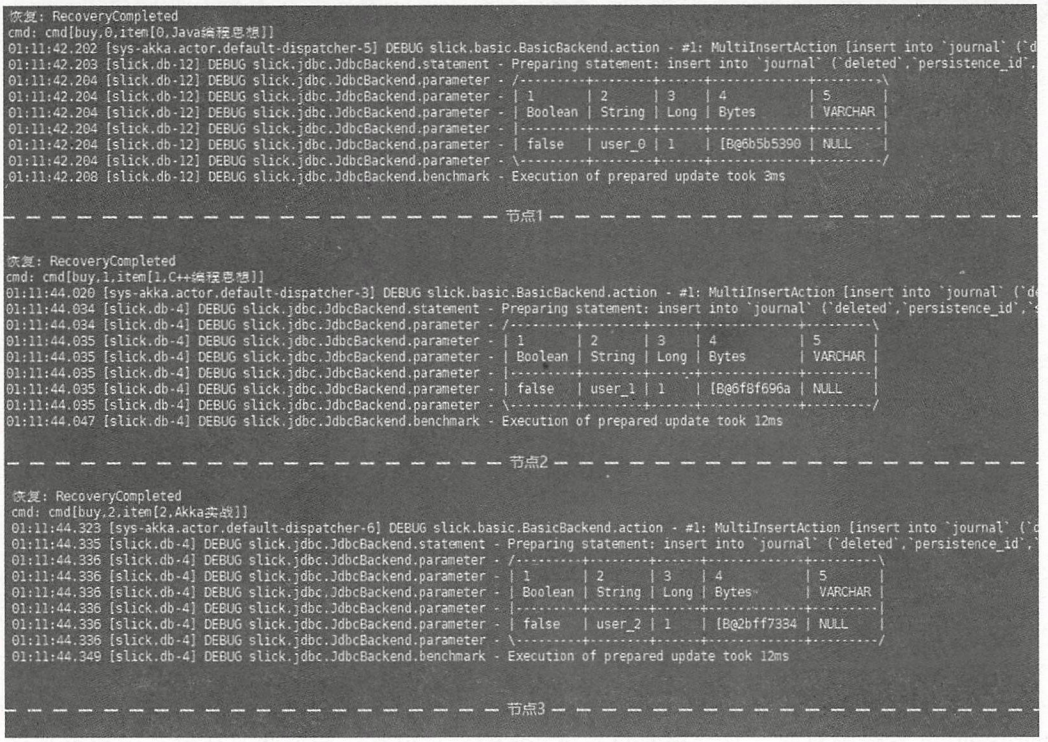



图 8-24　在 3 个节点上启动 ClusterSharding 并注册分片 Actor，处理“buy”命令



仅供非商业用途或交流学习使用

```
[root@localhost ~]# java -cp AkkaClusterDemo-1.0-allinone.jar com.akka.shard.eshop.StartService 192.168.115.135 2552 query
Actor[akka://sys/user/actorClient#2070591780] receive 0 cart:{0=item[0,Java编程思想]} from Actor[akka.tcp://sys@192.168.115.133:2550/system/sharding/userActor/0/0#1958324811]
Actor[akka://sys/user/actorClient#2070591780] receive 1 cart:{1=item[1,C++编程思想]} from Actor[akka.tcp://sys@192.168.115.135:2550/system/sharding/userActor/1/1#-143081330]
Actor[akka://sys/user/actorClient#2070591780] receive 2 cart:{2=item[2,Akka实战]} from Actor[akka.tcp://sys@192.168.115.136:2550/system/sharding/userActor/2/2#-1074213035]
```

图 8-25 在 2553 节点上通过 ShardRegion 访问分片 Actor，发送“query”命令

根据这两个结果图可以看出，我们确实已经将 Actor 分片到不同节点区域了。假如此时集群中加入了新的节点，或者移除（removed 状态）某个节点时，分片系统会自动平衡负载。

## 8.9 本章小结

本章我们主要了解了 Akka 集群的常规用法，它们包括：

- 创建并启动集群，感知所有节点的状态变化；
- 使用 akka-cluster 工具查询集群状态，或操作集群节点（比如将某个节点加入集群或移出集群）；
- 使用集群路由将消息转发到集群中的某个目标，而你不需要知道目标在哪个节点上；
- 使用集群客户端在外部访问集群系统内注册过的 Actor；
- 使用集群分片将 Actor 均匀地分布在集群的各个节点上，这些 Actor 都由一个 entityId 标识，它们地址透明，你无须知道它在哪个节点上运行。为了保存每次事件产生的状态，我们需要使用持久化 Actor 来实现。

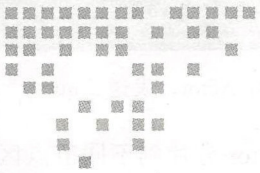
从目前来看，越来越多的互联网应用需要搭建自己的集群服务才能满足业务发展的需要，而 Akka 在 Actor 模型及远程功能的基础之上，提供了一系列构建完整集群服务所需的工具和配置，基本上达到了“开箱即用”的效果，大家在做技术选型时，可以考虑使用它作为集群服务的基础框架。

在下一章，我们将进入 Akka 最激动人心的主题之一：HTTP 服务。



仅供非商业用途或交流学习使用





Chapter 9 第 9 章

# HTTP 服务

随着互联网的高速发展，基于 HTTP 协议的 Web 服务越来越受到各个企业和开发者的重视，并且由于移动端（前端）的兴起，越来越多的前后端数据交互都会通过 HTTP 协议来实现。作为 Java 程序员，大家可能已经使用过 Java Web 体系下的某些技术开发过类似系统，比如 JSP、JSTL、EL、Servlet，或者其他成熟的开源框架，比如 Spring、Struts、Hibernate 等。

当我们使用这些技术搭建完系统后，需要将它部署在 Web 服务器（如 JBoss、Tomcat）中才能运行起来，这些服务器提供了多线程并发、I/O 等底层处理等，使得我们能把更多精力花在具体的业务实现上。而 Akka 在 Actor 和 Stream 的基础之上，也提供了一套用于构建 HTTP 服务的解决方案，即 Akka HTTP，它包含了服务端和客户端的实现，在运行时不需要借助其他 Web 服务器环境，为我们开发 Web 服务提供了另外一个选择。注意，Akka HTTP 并不是一款新的 Web 框架，它只是提供了一些处理 HTTP 协议的基础工具，我们可以在这个基础之上开发自己的 Web 框架。

在本章中，笔者将为大家介绍如何使用 Akka HTTP 来构建自己的 Web 服务。由于 Akka HTTP 涉及一些底层的处理，所以我们会先简单回顾一下 HTTP 协议的相关概念，然后再逐步介绍 Akka HTTP 相关的内容。

## 9.1 HTTP 协议

HTTP 协议主要用于客户端和服务端之间的通信，在 Web 应用中，客户端主要指浏览器。当我们打开浏览器并输入网址后，浏览器会向服务端发送 HTTP 请求，该请求中包含

仅供非商业用途或交流学习使用

了一些交互信息，我们称之为请求报文。请求报文中包括请求方式（比如 Get 或 Post）、访问目标、协议版本、header、实体内容等信息。当服务端接收到请求后，会向浏览器发出响应，同样，响应时也会携带报文信息，我们称之为响应报文。响应报文里面包含了协议版本、状态码、header、实体内容等信息。图 9-1 展示了访问 http://www.iteye.com 时的请求 – 响应过程，以及部分报文信息。

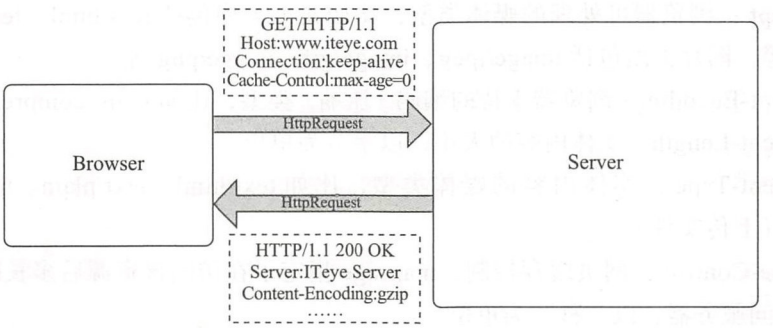



图 9-1　HTTP 的请求（HttpRequest）和响应（HttpResponse）

## 9.1.1　header 信息

在请求和响应的报文里都包含一些 header 信息，它们是客户端和服务端处理请求和响应时必要的数据。这些数据可以通过浏览器进行查看，比如 Chrome 浏览器、按下 F12 键进入开发者工具，点击 Network 选项卡，选择某一个 URL，点击后即可在右边看到本次请求相关的信息。图 9-2 展示了通过 Chrome 开发者工具查看到的 header 信息。

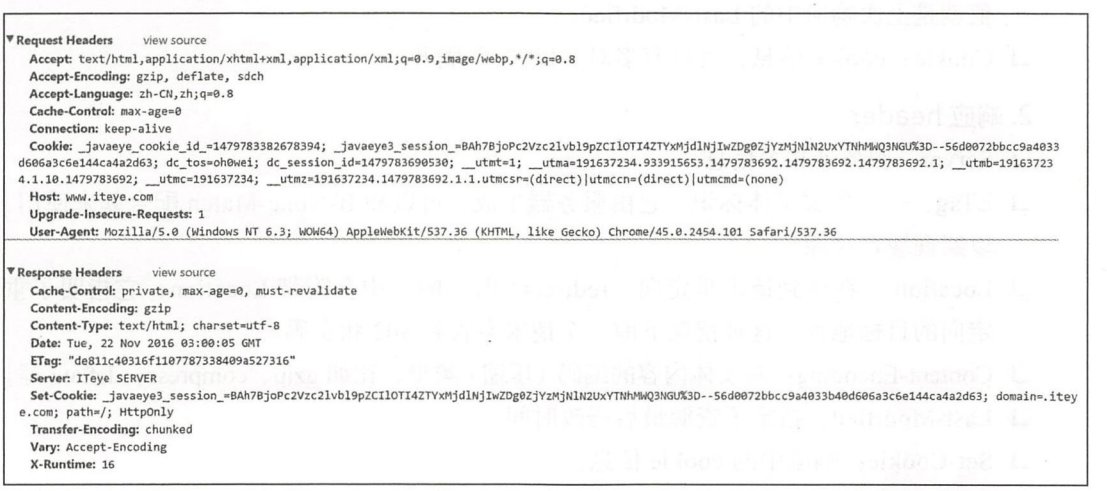



图 9-2　请求和响应中的 header 信息

仅供非商业用途或交流学习使用

这些 header 信息都是由字段名 : 字段值组成的，每一个 header 都有自己的含义。HTTP 的 header 包括通用 header、请求 header、响应 header 和实体 header 这四类，下面笔者将大致按照请求 header、响应 header 来进行介绍，涉及通用 header 或实体 header 的也将包含在内，不再做分开介绍。

### 1. 请求 header

- Accept：浏览器可处理的媒体类型，比如文本类型包括 text/html、text/plain、text/css 等，图片类型包括 image/jpeg、image/gif、image/png 等。
- Accept-Encoding：浏览器支持的编码（压缩）类型，比如 gzip、compress、deflate 等。
- Content-Length：实体内容的大小，以字节为单位。
- Content-Type：实体内容的媒体类型，比如 text/html、text/plain、multipart/form-data（上传文件）。
- Cache-Control：网页缓存控制，max-age 指定了在访问该资源后多长时间内不会再次访问服务器，以“秒”为单位。
- Connection：是否开启持久连接，Keep-Alive 表示开启，Close 表示关闭。
- Host：请求的服务器域名和端口号。
- User-Agent：客户端信息，包括操作系统、浏览器等信息。
- Referer：请求的来源地址，通过它服务端可以得知请求是从哪个页面发过来的。
- If-Modified-Since：通常和 Last-Modified 配合起来使用。服务器可以给客户端发送一个 Last-Modified 以指定资源的最后更新时间。下一次请求时，请求报文会附带 If-Modified-Since。如果在 If-Modified-Since 之后没有更新过，会返回 304 状态码，让客户端使用缓存；假如更新过，则从服务器获取最新数据。If-Modified-Since 的值就是上次响应中的 Last-Modified。
- Cookie：cookie 信息，可以有多对，以“;”隔开。

### 2. 响应 header

- Server：服务器相关信息。
- ETag：一个资源实体标识，它由服务器生成，可以和 If-None-Match 配合起来使用，以实现缓存功能。
- Location：在遇到请求重定向（redirect）时，响应中会附带 Location，它指明了重定向的目标地址，这种情况下前一个请求会收到 302 状态码。
- Content-Encoding：对实体内容的编码（压缩）类型，比如 gzip、compress、deflate 等。
- Last-Modified：指定了资源最后修改时间。
- Set-Cookie：响应中的 cookie 信息。

上面只是列举了部分常见的 header，更多 header 信息大家可自行参考 HTTP 协议的相关资料。

仅供非商业用途或交流学习使用

### 9.1.2　状态码

状态码在服务端用于描述请求结果。每个状态码对应一个描述字符串。状态码一共有 5 种，如表 9-1 所示。

表 9-1　HTTP 状态码简介

| 状　态　码 | 描　　述 | 示　　例 |
| --- | --- | --- |
| 1xx | 请求正在（继续）被处理 | 比如 100，表示客户端继续发送请求 |
| 2xx | 请求被成功接收 | 比如 200，表示请求已经成功 |
| 3xx | 请求重定向、资源未修改 | 比如 302，表示重定向；304，表示资源未被修改 |
| 4xx | 请求错误，或找不到资源 | 比如 400，表示错误的请求格式；404，表示找不到资源 |
| 5xx | 服务器内部发生错误 | 比如 500，表示服务端程序异常；502，表示网关错误 |

## 9.2　Akka HTTP

Akka 提供了构建完整 HTTP 服务所需的各个基础组件，在真正动手开发前，我们先来了解这些组件的基本含义及用法，这有助于我们对后面内容的理解。

- HttpRequest：表示一个 HTTP 请求，根据 HTTP 协议，它包括各种请求方法（比如 get、post、put、delete 等）、header、实体内容（entity）等数据。HttpRequest 对象可以通过 HttpRequest.create 方法创建。
- HttpResponse：表示一个 HTTP 响应，根据 HTTP 协议，它包括各种状态码（比如 200，304 等）、header、实体内容（entity）等数据。HttpResponse 对象可以通过 HttpResponse.create 方法创建。
- HttpEntity：表示请求和响应中携带的数据。HttpEntity 对象有多种类型，比如 HttpEntity.Strict、HttpEntity.Chunked。前者是最简单的一种类型，一般在已知 Content-Length，并且数据量不大时使用；后者表示使用 chunked（分块）的方式进行数据传输，它会将 header 中的 Transfer-Encoding 设置为 chunked。
- Header：当 Akka HTTP 服务接收到请求后，会解析所有的 header，比如前面介绍的 Content-Length、Content-Type、User-Agent 等。

### 9.2.1　配置依赖

为了保证核心模块的易读性和可维护性，Akka HTTP 提供了两个级别的 API 来实现服务端，低级别的 API 主要实现基础的功能，比如连接管理、请求和响应、解析 Header 等；高级 API 则实现一些上层的功能，比如路由、文件处理等。这两套 API 分别在 akka-http-core、akka-http 这两个模块里，它们可以混在一起使用，在使用前需要在工程中加入以下 maven 依赖：

仅供非商业用途或交流学习使用

```
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-http-core_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-http-experimental_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
```

### 9.2.2 HTTP 服务端

由于 Akka HTTP 建立在 Stream 的基础之上，所以它的很多操作都被抽象成 Stream 的各个组件的数据流操作。当一个 Akka HTTP 服务启动后，它会从 Source 中读取连接对象并交给 Sink 处理，得到这些连接对象后，再将其分配到 Flow 或者 Function 中以便进行 HttpRequest 和 HttpResponse 相关的操作。下面我们先看看怎样使用低级别的 API 构建一个 HTTP 服务：

```
ActorSystem system = ActorSystem.create("sys");
Materializer mat = ActorMaterializer.create(system);
Source<IncomingConnection, CompletionStage<ServerBinding>> bindSource =
Http.get(system).bind(ConnectHttp.toHost("localhost", 8089),mat);
CompletionStage<ServerBinding> bindFuture = bindSource.to(
    Sink.foreach(conn -> {
      //处理请求...
System.out.println("来自"+conn.remoteAddress()+"的访问!");
})).run(mat);
```

如上述代码所示，我们首先定义了 Materializer 对象，它会在内部创建 Actor 来运行 Stream。然后通过 HTTP 对象绑定地址和端口，绑定成功后会得到一个含有 IncomingConnection 的 Source 对象，该对象是服务端后续处理请求（连接）的起点。IncomingConnection 表示一次请求的连接，当客户端访问时，会创建该对象。接着，我们使用常规的流的操作将 Source 连接到 Sink 中进行处理，在 Sink.foreach 中，我们可以得到前面创建的 IncomingConnection 对象。到这里，一个初级的 HTTP 服务算是开发完毕了。启动这个程序后，我们就可以在浏览器中输入 http://localhost:8089/ 进行访问了。

### 9.2.3 请求和响应

一个常规的 HTTP 服务必然包括接收请求、处理请求、响应请求这几个处理过程，而前面的示例只能满足第一个，即“接收请求”，那么我们该怎么实现第二、第三个呢？

在前面的示例中，我们在 Sink.foreach 里得到了每个来访的 IncomingConnection 对象，该对象拥有一系列以 handleWith 为开头的方法可用于处理请求，比如 handleWith、

handleWithSyncHandler 等；

- handleWith 方法需要一个 Flow<HttpRequest, HttpResponse, Mat> 组件来定义处理逻辑；
- handleWithSyncHandler 方法需要一个 Function<HttpRequest, HttpResponse> 函数来定义处理逻辑。

以 handleWithSyncHandler 为例，我们可以先定义一个 Function，然后实现其 apply 方法，该方法用于处理请求并返回响应：

```
// 处理请求
    Function<HttpRequest, HttpResponse> processRequest = new Function<HttpRequest,
HttpResponse>() {
    // 定义404响应对象
    HttpResponse resp_404 = HttpResponse.create().withStatus(404).
withEntity("找不到页面...");
    @Override
    public HttpResponse apply(HttpRequest request) throws Exception {
        // 获取请求path
        String path = request.getUri().path();
        if (path.equals("/")) {
            // 响应HTML
            return HttpResponse.create()
.withEntity(ContentTypes.TEXT_HTML_UTF8,"<font color='red'>欢迎来到首页
</font>");
        } else if (path.equals("/items")) {
          Query query=request.getUri().query();
          String itemId = query.get("itemId").orElse("not found");
          return
            HttpResponse.create().withEntity("查找商品"+itemId+"!");
        }else if(path.equals("/redirect")){
            //重定向
            Location locationHeader = Location.create("http://localhost:8089/");
            return HttpResponse.create()
                .withStatus(StatusCodes.FOUND).addHeader(locationHeader);
        }else {
            return resp_404;
        }}
};
```

然后将其传入 handleWithSyncHandler 方法：

```
conn.handleWithSyncHandler(processRequest,mat);
```

假如你要使用 handleWith 方法，则需要创建一个 Flow，然后做 map 处理：

```
conn.handleWith(Flow.of(HttpRequest.class).map(processRequest),mat);
```

现在我们来看看 processRequest 到底做了哪些工作。

当客户端来访时，它会进入 processRequest 的 apply 方法，该方法将请求对象作为参数传入，等到处理结束时它需要一个 HttpResponse 对象作为响应。由于客户端可能会访问不

同的 URL，所以我们先通过 HttpRequest 得到用户的目标 URL，然后匹配到不同的处理流程，进而返回不同的响应对象。

响应对象通过 HttpResponse.create() 创建，假如需要返回不同的状态码，可以调用 withStatus 方法进行指定，这里你既可以指定为数字，也可以指定为 StatusCodes 下的相关静态变量，比如：

```
// 创建一个状态码为200的响应对象
HttpResponse resp_ok = HttpResponse.create().withStatus(StatusCodes.OK);
// 创建一个带有错误提示的404响应对象
HttpResponse resp_404 = HttpResponse.create()
.withStatus(StatusCodes.NOT_FOUND).withEntity("找不到页面...");
```

当用户访问的地址不存在时，直接给对方返回一个 404 响应，并提醒“找不到页面……”。

当用户访问“/”时候，我们希望能响应 HTML 给对方，此时需要调用 withEntity 方法来指定实体内容，对于 HTML 类型的实体，需要传入 ContentTypes.TEXT_HTML_UTF8 参数，将响应 header 里的 Content-Type 设置为“text/html”。该值默认为“text/plain”，它会让浏览器按原样输出文本，而不会渲染 HTML 效果。withEntity 里面还支持传入 HttpEntity 对象，该对象使用 HttpEntities 创建，比如：HttpEntities.create("Hello Entity")，假如是 HTML，则需要在第一个参数中传入 ContentTypes.TEXT_HTML_UTF8。

当用户访问“/items”时，假如地址后面含有查询字符串（即？后的参数），则可以通过 Query 对象来获取，假如没有该参数，则将 itemId 变量设置为 not found。

当用户访问“/redirect”时，由于这里创建了一个重定向的响应，所以服务器会返回 302，并马上跳转到 http://localhost:8089/，如图 9-3 所示。

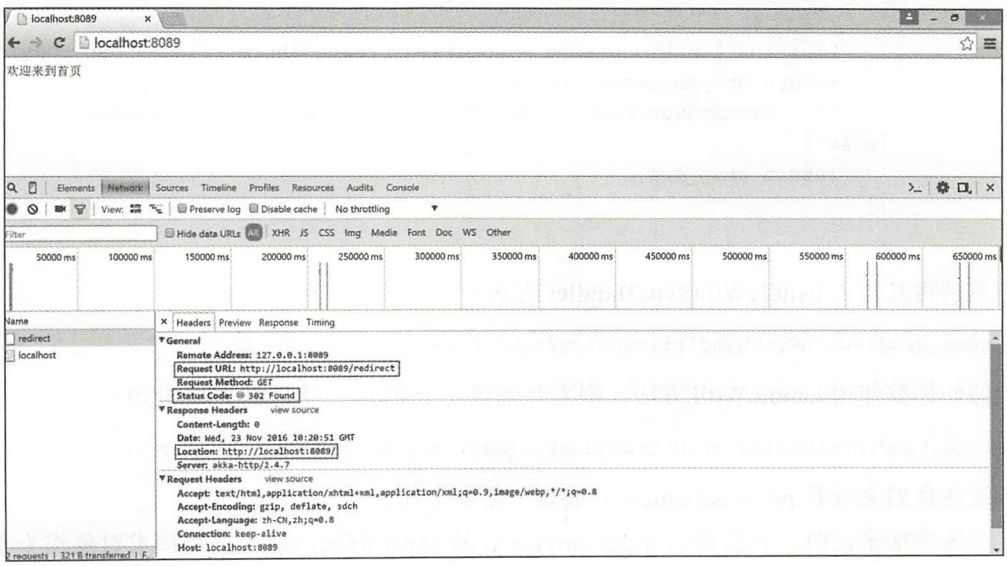



图 9-3　重定向效果图

仅供非商业用途或交流学习使用





### 9.2.4 HTTP 客户端

一般情况下，我们会使用浏览器作为客户端来访问 Web 服务，但是在有些项目中需要直接通过应用程序请求一个 Web 资源（比如爬虫），对此，Akka HTTP 提供了客户端 API 的直接支持。

使用客户端 API 的第一步是打开 HTTP 连接，该连接由 OutgoingConnection 表示。打开连接后，再借助 Source 向其发送请求，正常情况下，此时会返回一个响应，即 HttpResponse 对象，该对象被封装在 CompletionStage 中以便进行异步读取。下面是一个简单的示例：

```
Flow<HttpRequest, HttpResponse, CompletionStage<OutgoingConnection>> connFlow =
Http.get(system).outgoingConnection(ConnectHttp.toHost("localhost", 8089));
CompletionStage<HttpResponse> respFuture =
Source.single(HttpRequest.create("/shoppingcar")).via(connFlow).runWith(Sink.
<HttpResponse> head(), mat);
respFuture.thenAccept(response->{
    response.entity().getDataBytes().runWith(Sink.foreach(content->{
        System.out.println(content.utf8String());
    }), mat);
});
```

代码中的 HttpRequest.create 方法指定了具体的访问地址，所以实际上，该程序最终会访问 http://localhost:8089/shoppingcar。请求时你可以附加一些 header 信息，比如 cookie:

```
HttpRequest.create("/shoppingcar").addHeader(Cookie.create("sid","12345"))
```

当请求成功后，对方会将响应数据包装在 HttpResponse 的 entity(ResponseEntity) 里。为了能将这些数据取出来，我们需要调用 getDataBytes() 方法得到一个 Source 对象，然后通过流的方式来取值，由于这些数据是 ByteString 类型的，所以要将其转化成可读的 String 类型。

### 9.2.5 使用 Routing DSL

在前面笔者介绍过，Akka 提供了两个级别的 API 来实现 HTTP 服务，前面我们使用了 low-level 的 API，它实际上已经能满足基本的需求了，只不过看起来比较臃肿，可读性也比较差。Akka 提供的 high-level 的 API 则比较简洁且表现力极强，我们称之为“Routing DSL”。

“Routing DSL”的核心是 Route 和 Directive。Route 是一个用于匹配和处理请求的函数，它最终会返回一个 RouteResult 以描述可能的结果，而 Directive 主要用于创建和组装多个 Route。Akka HTTP 已经内置了很多 Directive，使用它们可以实现大部分常见功能，比如解析请求数据（如 cookie、heade）、处理或转换响应数据等。

下面，我们先看一个完整的示例再来分析“Routing DSL”的使用过程：

```
public class App extends AllDirectives {
    public void startApp() {
        ActorSystem system = ActorSystem.create("sys");
```

仅供非商业用途或交流学习使用





```
        Materializer mat = ActorMaterializer.create(system);
Flow<HttpRequest, HttpResponse, NotUsed> routeFlow = getRoute().flow(system, mat);
        CompletionStage<ServerBinding> binding =
Http.get(system).bindAndHandle(routeFlow, ConnectHttp.toHost("localhost", 8080),mat);
    }
    public Route getRoute() {
        return get(() -> route(
            path("index",()->complete("欢迎来到首页")),
            path("books",()->complete("书籍列表")),
            path("book",()->parameterOptional("bookid", bookid -> {
                String bid = bookid.orElse("-1");
                return complete("查询的数据是: "+bid);
            })),
            path("shoppingcar",()->{
                HttpResponse resp=HttpResponse.create()
.withEntity(ContentTypes.TEXT_HTML_UTF8,"<font color='red'>购物车列表</font>");
                return complete(resp);
            })
    ));
}
    public static void main(String[] args) {
        App app=new App();
        app.startApp();
    }
}
```

在这个例子中，我们创建了一个本地的 HTTP 服务，它对外提供多个访问路径，每次访问都以 complete 方法结束，该方法会按照指定的 Content-Type 和实体内容响应客户端。大家会发现，这段代码的可读性非常强，即使不深入细节也能知道它是做什么的。

使用“Route DSL”最重要的一步是：创建用于处理请求的 Route。在这个过程中需要使用大量不同的 Directive，为了方便起见，我们可以直接定义一个类来继承 AllDirectives，这样可获取所有 Directive。

创建 Route 的过程在 getRoute 方法里面，它使用 MethodDirectives.get 创建了一个用于处理所有 get 请求的 Route，假如你希望同时能处理其他类型的请求，可以将 get 处理去掉，直接返回 route(...)。

route 里面可以传入多个 path 方法，以处理不同的访问路径。path 方法来自于 PathDirectives，当请求被匹配成功后，底层会调用它的回调函数（Supplier 类型）来处理这个请求，该函数需要返回一个 Route 对象。

为了响应客户端，需要调用 complete 方法来完成本次处理，它会返回一个 Route 对象。当你在 complete 方法中直接传入普通字符串时，它会响应“text/plain”类型的数据，假如你要返回网页内容，则需要明确指定 Content-Type。

在处理 get 请求时，一个很常见的操作是获取查询字符串参数，此时我们可以使用 parameter 或 parameterOptional 两个 directive 函数来处理，这两个 directive 函数都来自于

仅供非商业用途或交流学习使用





ParameterDirectives。

这个示例仅仅实现了对 get 请求的简单处理，为了完成复杂的 Web 应用，我们还需要借助于其他 Directive，接下来笔者将介绍一些常见 Directive 的用法。

### 9.2.6 常用 Directive

#### 1. HeaderDirectives

HeaderDirectives.headerValueByName 可以通过名字获取 header 信息，比如浏览器相关信息：

```
path("category", () -> headerValueByName("User-Agent",u->complete("获取header:"+u)))
```

#### 2. PathDirectives

PathDirectives. pathSingleSlash 可以匹配首页路径，即“/”：

```
pathSingleSlash(()->complete("默认首页"))
```

PathDirectives.path 可以匹配指定路径，比如“/index”：

```
path("index", () -> complete("欢迎来到首页"))
```

#### 3. ParameterDirectives

ParameterDirectives.parameter 可以获取查询字符串中的参数，第一个参数（name）表示字符串参数名：

```
path("searchBook",()->parameter("name",name->complete("查询到的书: " + name)))
```

ParameterDirectives.parameterMap 能以 map 的格式获取所有查询字符串参数：

```
path("searchBookBy",()->parameterMap(p->{
    return complete("查询的数据是: "+p.get("name")+","+p.get("price"));
}))
```

大家可以输入 searchBookBy?name=akka&price=100 这样的 URL 进行测试。

#### 4. FormFieldDirectives

FormFieldDirectives.formFieldMap 能以 map 的格式获取所有 method 为 post 的 form 表单域的值：

```
path("addBook",()->formFieldMap(fm->{
    //获取表单里面的数据
    return complete("提交的数据是: "+fm);
    }))
```

#### 5. CookieDirectives

CookieDirectives.cookie 可以根据名字来获取 cookie 值：

仅供非商业用途或交流学习使用



```
path("getcookie",()->cookie("name", name->complete("cookie的数据是: "+name)))
```

CookieDirectives.setCookie 可以设置一个 cookie：

```
path("setcookie",()->setCookie(HttpCookie.create("sid","user_123"),()->comple
te("设置cookie")))
```

### 6. FileUploadDirectives

FileUploadDirectives.uploadedFile 用于处理文件上传，假如上传的文件在写入硬盘时出错，则请求失败。假如同一个名字对应多个文件，那么会取第一个，其他的丢弃，下面是示例代码：

```
path("uploadImg",()->uploadedFile("myimg",(FileInfo info,File inputFile)->{
    File outputFile=new File("C:\\jeework\\HttpServer\\doc\\test.jpg");
    //复制到目的地?
    FileUtils.writeToFile(inputFile, outputFile);
    return complete("上传成功"+info.getFileName()+""+info.getContentType());
    }))
```

### 7. RouteDirectives

RouteDirectives.redirect 可以产生一个重定向响应，当用户访问目标路径时（如“/redirect_to”），会直接跳转到指定的目标页面（“/index”）：

```
path("redirect_to",()->redirect(Uri.create("/index"), StatusCodes.FOUND))
```

实际上，我们之前使用的 route、complete 等方法也是由 RouteDirectives 提供的，这里不再给出示例。

## 9.3 Akka WebSocket

通常情况下，HTTP 服务器都是比较“被动”的，只有在浏览器发送请求之后，它才会将数据响应出来，本身没有主动推送数据的能力。而在诸如一些 IM、实时监控等应用中，我们往往需要服务端能够即时地将信息推送给浏览器，那么这类功能该怎么实现呢？

在早期，由于 HTTP 协议自身的限制，我们在开发这类应用时，大都会采用“Ajax 轮询”或者 Comet 技术来实现。“Ajax 轮询”的意思是在一定的时间间隔内向服务端发起一次 HTTP 请求并获取最新数据，这种做法比较简单，但是间隔时间不好控制，间隔太短会增大服务器的压力，而间隔太长则会降低时效性。Comet 技术主要是指在长连接的基础之上，通过“长轮询”来模拟浏览器与服务端的双向通信，以实现服务器的推送功能，这种做法也需要服务端强有力的支持。

而现在，我们可以有一个更好的选择，那就是 WebSocket。WebSocket 是 HTML5 时代兴起的一种通信协议，它在 Tcp 的基础之上，提供了浏览器和服务器之间的全双工通信的能力，从源头上解决了服务端“被动性”的问题。

仅供非商业用途或交流学习使用



### 9.3.1 WebSocket 协议

在建立 WebSocket 连接时，浏览器和服务器只需要进行一次握手，即可快速建立一条双向消息通道，接着就可以即时传递数据了。在发起握手请求时，它仍然需要借助于 HTTP 来完成，和正常 HTTP 请求不同的是，它携带了部分与 WebSocket 相关的 header 信息，如图 9-4 所示。

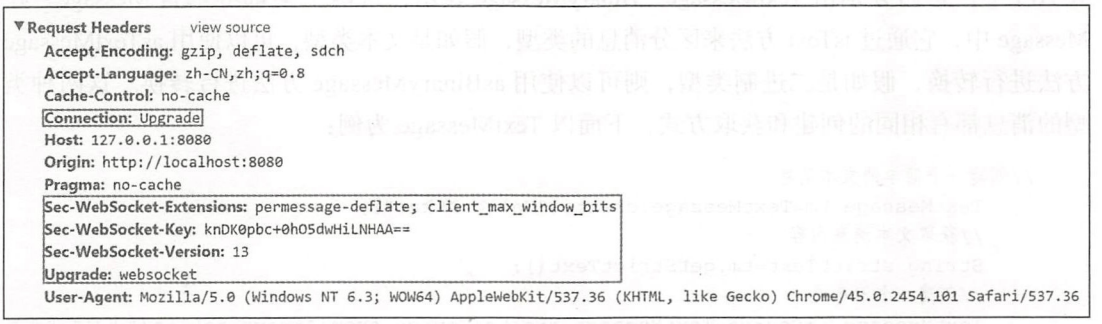



图 9-4 WebSocket 请求信息（图截自 Chrome）

当请求成功后，服务端会返回一个握手成功的响应，看起来如图 9-5 所示。

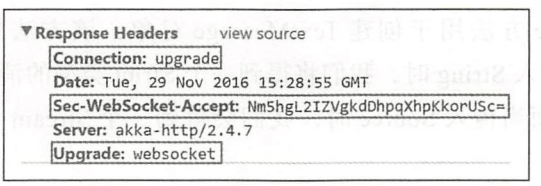



图 9-5 WebSocket 响应信息（图截自 Chrome）

现在，我们来看看请求和响应中与 WebSocket 相关的 header 信息。在浏览器发起握手请求时，它携带的 Connection:Upgrade 和 Upgrade:websocket 表明当前是一个 WebSocket 请求，Sec-WebSocket-Key 是一个 base64 编码的随机字符串，服务端会根据该字符串做加密处理，然后作为 Sec-WebSocket-Accept 这个 header 响应出来。请求 header 中的 Sec-WebSocket-Extensions 不是必须的，假如存在则表示协议扩展或增强。Sec-WebSocket-Version 表示当前协议版本。服务端除了返回 Sec-WebSocket-Accept 之外，还会返回 Connection:Upgrade 和 Upgrade:websocket，这表明服务端已经成功切换协议了，也就是说，接下来将会以 WebSocket 协议的模式进行数据传输。

### 9.3.2 Akka WebSocket

要使用 WebSocket，离不开客户端（浏览器）和服务端的支持。对于浏览器来讲，HTML5 已经提供了非常易用的 API，编写一个基础应用的核心代码不超过 10 行。对于服

仅供非商业用途或交流学习使用



务端来讲，目前主流的 Web 服务器都已经支持 WebSocket 协议了，比如 Tomcat、Jetty。而 Akka HTTP 模块在现有模型的基础之上，也提供了一套基于流的 WebSocket API，下面我们看看 Akka WebSocket 主要有哪些内容。

### 1. 消息类型

通过 WebSocket 传输的数据（也称消息）既可以是 unicode 文本，也可以是二进制流，在 API 中，它们分别由 TextMessage、BinaryMessage 表示，这两个类都继承自 Message。在 Message 中，它通过 isText 方法来区分消息的类型，假如是文本类型，可以使用 asTextMessage 方法进行转换，假如是二进制类型，则可以使用 asBinaryMessage 方法进行转换。这两种类型的消息都有相同的创建和获取方式，下面以 TextMessage 为例：

```
//创建一个简单的文本消息
    TextMessage tm=TextMessage.create("Hello Akka!");
    //获取文本消息内容
    String strictText=tm.getStrictText();
    //创建一个消息流
    TextMessage tmSource=TextMessage.create(Source.from(Arrays.asList("上海","北京
        ","南京")));
    //获取消息流Source
    Source<String,?> streamText=tmSource.getStreamedText();
    streamText.runWith(Sink.foreach(System.out::println),mat);
```

TextMessage.create 方法用于创建 TextMessage 对象，该方法里可以传入两类参数：String 和 Source。当传入 String 时，我们将得到一个 Strict 类型的消息，它表示一种比较确认的、完整的消息。而当传入 Source 时，我们会得到一个 Stream 类型的消息，该消息即 Source 中产生的数据。

当服务端接收到来自客户端的消息时，我们不能想当然的认为这些消息必然属于 Strict 或者 Stream。假如使用 getStrictText 来接收一个 Stream 消息会引发异常，所以一般情况下，我们会先调用 TextMessage 的 isStrict 方法来判断消息的类型，然后分别做处理。

### 2. 请求和响应

在 Akka 里面，WebSocket 的请求和响应仍然使用 HttpRequest 和 HttpResponse 来表示，也就是说，当浏览器发起一个 WebSocket 请求时，我们可以使用之前的 HTTP 的方式来接收这个请求。而在处理请求时，我们需要定义一个 Flow<Message,Message,?> 类型的组件，通过泛型可以看出，这个 Flow 能接收到来自浏览器的消息，也能够产生将要发送给浏览器的消息。和 HTTP 类似，响应 WebSocket 请求时也需要返回一个 HttpResponse 对象。下面这段代码实现了接收并响应一次 WebSocket 请求的基本过程：

```
if (path.equals("/ws_demo")) {
    Flow<Message,Message,NotUsed> flow=Flow.<Message>create().map(msg->{
    TextMessage in_msg=msg.asTextMessage();
    TextMessage out_msg=TextMessage.create("你好啊,"+in_msg.getStrictText());
```

仅供非商业用途或交流学习使用



```
        return out_msg;
        });
        return WebSocket.handleWebSocketRequestWith(request,flow);
    }
```

如上述代码所示，当服务端接收到一个请求后，我们通过 WebSocket.handleWebSocket-RequestWith 方法返回一个 WebSocket 响应。该方法会通过查找 HttpRequest 里的 Upgrade-ToWebSocket（header 信息，后面会继续提到）来识别当前请求是否是一个有效的 WebSocket 请求。假如是，那么会调用 Flow 来处理这次请求；假如不是，则直接返回状态码为 400 的响应。在 Flow 里，我们通过 map 函数将浏览器发送过来的消息取出来，然后将其转换成文本类型（TextMessage），假如浏览器发送的是文件流数据，那么应该调用 asBinaryMessage 方法将其转换成二进制类型（BinaryMessage）。map 函数返回的 TextMessage 将会被作为响应数据推送给浏览器，此时浏览器可以即时得到这个数据。

细心的读者可能会发现，UpgradeToWebSocket 这个 header 貌似并不存在（参考图 9-4 所示的浏览器截图），那么这里怎么能得到它呢？为了验证这个问题，我们可以在服务端使用以下代码打印出本次请求所有相关的 header：

```
request.getHeaders().forEach(System.out::println);
```

运行的结果显示，请求里确实存在 UpgradeToWebSocket。实际上，这个 header 并不是浏览器直接发送的，而是在服务端接收请求后自动附加上去的，一般情况下我们无须关心这个细节。

前面这段代码仅仅演示了 Akka WebSocket 中最核心的 API，目的是让大家对 WebSocket 的使用有一个大致的了解。由于这里还没有介绍浏览器端（HTML5 WebSocket API）的操作，所以它尚不具备完整功能，接下来，笔者将通过一个聊天室的示例程序来演示如何编写一个完整的拥有“前后端”即时交互能力的 WebSocket 应用。

### 9.3.3　WebSocket 应用之聊天室

之所以选择聊天室作为示例项目，主要是因为它最能完整体现 WebSocket 即时通信的能力。在这个示例中，我们将要实现的功能是：当用户进入聊天室后，可以发送普通文本信息和上传图片，此时聊天室中的其他成员可以即时收到这些消息。当某个用户离开聊天室后，服务端可以收到通知并更新客户端列表。

从功能描述上可以看出，服务端需要能够给所有用户推送消息，很显然，前面那段示例代码并不能完全实现这个功能，它只能对当前用户推送消息。一个比较好的做法是，当一个用户进入聊天室后，我们就新建一个 Actor 来代表这个用户，后面所有对该用户的推送都会经过这个 Actor，我们把这个 Actor 取名为 User。同时，我们可以引入一个 ChatRoom 的 Actor，它主要做两件事：

1）管理所有 User，即当某个用户进入聊天室时，将 User 注册到用户列表；当用户离

仅供非商业用途或交流学习使用



开聊天室时（断开连接），将 User 从用户列表中移除。

2）统一接收来自用户的消息，并将消息分发给所有 User，让 User 最终推送给浏览器。

现在的问题是：我们该怎样让来自用户的消息转入到 ChatRoom 里面，然后又怎样通过 User 向浏览器推送消息呢？

在前面我们了解到，处理一个 WebSocket 请求需要借助于 Flow 组件，这个组件拥有“输入 – 输出”的接口，刚好可以对应到“请求 – 响应”的模型。Flow 组件本身是可以由 Sink 和 Source 来构建的，Sink 用于描述输入，Source 用于描述输出。在使用 Sink 时，我们可以将其关联到一个 Actor，让这个 Actor 来接收请求数据。按照前面的设想，我们可以将此 Actor 作为 ChatRoom。而在使用 Source 时，我们可以将其内部的 Actor 取出来，根据 Stream 的机制，当我们向这个 Actor 发送消息时，它会将这些消息发布到下游的流中，在当前环境下，下游即用户。

现在，我们可以大致梳理出这个聊天室的基本架构，如图 9-6 所示。

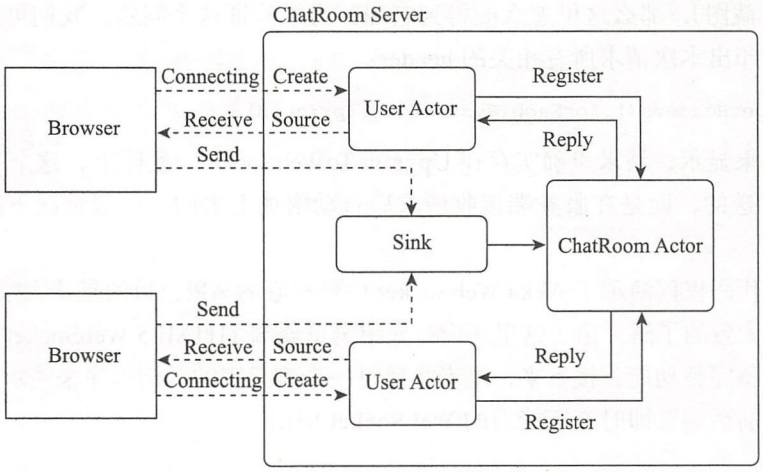



图 9-6　聊天室基本架构

接下来，我们用 Akka WebSocket 来实现下这个聊天室。

### 1. 服务端实现

首先，我们需要定义一个 HTTP 服务，它监听某个端口，等待外部的连接。为了简化整个程序，我们可以定义一个单独的 class 来做处理，并且将一些公共资源（如 ActorSystem、Materializer）提取出来。下面是这个 class 的完整代码：

```
public class WsChatRoomApp {
    private ActorSystem system;
    private Materializer mat;
    //聊天室Actor
    private ActorRef chatRoom;
    /**
```

仅供非商业用途或交流学习使用



```
     * 启动Web服务
     */
    public static void main(String[] args) {
        WsChatRoomApp chat = new WsChatRoomApp();
        chat.startApp();
    }
    public void startApp() {
        system= ActorSystem.create("sys");
        mat = ActorMaterializer.create(system);
        Source<IncomingConnection, CompletionStage<ServerBinding>> bindSource =
            Http.get(system).bind(ConnectHttp.toHost("127.0.0.1", 8080), mat);
CompletionStage<ServerBinding> bindFuture = bindSource.to(Sink.foreach(conn ->
{
        conn.handleWithSyncHandler(processRequest, mat);
        })).run(mat);
        chatRoom = system.actorOf(Props.create(ChatRoom.class,mat), "chatRoom");
    }

    Function<HttpRequest, HttpResponse> processRequest = new Function<HttpRequest,
        HttpResponse>() {
    @Override
    public HttpResponse apply(HttpRequest request) throws Exception {
    // 获取请求uri
    Uri uri = request.getUri();
    // uri.path()获取请求地址
    if (uri.path().equals("/")) {
    // FileUtils工具类仅用来读取文件内容，大家可自行实现，这里不再给出代码
    String index = FileUtils.readHtml("index.html");
    return HttpResponse.create().withEntity(
    ContentTypes.TEXT_HTML_UTF8, index);
    } else if (uri.path().equals("/ws_demo")) {
    //连接聊天室，新建一个UserActor进行处理
    ActorRef userActor = system.actorOf(Props.create(User.class,chatRoom));
    Sink<Message, NotUsed> inMsg = Sink.actorRef(chatRoom,new Notice(userActor));
    Source<Message, NotUsed> outMsg = Source.<Message> actorRef(100, Overflow
        Strategy.fail())
        .mapMaterializedValue(outRef -> {
    outRef.tell(TextMessage.create("<font color='red'>您已进入聊天室...</font>"),
        ActorRef.noSender());
    userActor.tell(new Conn(), outRef);
    return NotUsed.getInstance();
    });
    Flow<Message, Message, NotUsed> flow = Flow.fromSinkAndSource(
    inMsg, outMsg);
    return WebSocket.handleWebSocketRequestWith(request, flow);
    } else {
    return HttpResponse.create().withStatus(404)
    .withEntity("找不到页面...");
    }
}};}
```

仅供非商业用途或交流学习使用



就如前面所讲，这个服务既可以接收 HTTP 请求，也可以接收 WebSocket 请求，底层会通过读取 Header 信息来识别请求的类型。

处理请求的核心逻辑在 processRequest 方法里面。当用户访问首页（即“/”）时，我们使用 FileUtils 工具类来读取 index.html，并将其响应给用户。index.html 就是聊天室的主页，该页面会请求“/ws_demo”以发起 WebSocket 连接，当连接成功后，用户就可以发送和接收消息了。这个页面涉及 HTML5 WebSocket API 的使用，笔者将在本节后面对其进行讲解，这里暂时不给出源码。

当用户连接进入聊天室后，我们首先创建了一个 User Actor，它主要负责关联用户，承担着给用户推送消息的责任。紧接着，我们创建了一个 Sink 以接收来自客户端的请求，Sink.actorRef 指定了 ChatRoom 作为消息的统一接收者，当超时或连接中断时，程序会发送 Notice 信息给 ChatRoom，以提醒它关闭对应的 User Actor。为了能给用户推送消息，我们创建了一个 Source 对象，然后通过 mapMaterializedValue 方法将内部 Actor（即 outRef）提取出来，这个 Actor 此时可以看成是与用户的一个消息通道，现在，只要给该 Actor 发送消息，用户就能即时的收到这个消息。如代码所示，这里首先向用户推送了“您已进入 ...”的初始消息，接着，我们给当前的 UserActor 发送了一个 Conn 的指令，表明连接已就绪，此时把 outRef 作为 sender 的主要原因是：我们需要在 UserActor 里面绑定对外的消息通道。这样的话，一个 UserActor 才能真正代表一个用户。最后，我们通过 Flow.fromSinkAndSource 方法将 Sink 和 Source 连接起来，形成一个新的 Flow 并将其作为处理请求的 handler。

由于这个聊天室支持文本和二进制两种消息类型，所以当消息进入 ChatRoom 后，我们需要先判断类型，然后分别处理。下面是 ChatRoom 和 User 的示例代码：

```
/**
 * 聊天室
 */
class ChatRoom extends UntypedActor {
    private List<ActorRef> clientActors=new ArrayList<ActorRef>();
    private Materializer mat;
    public ChatRoom(Materializer mat) {
      this.mat=mat;
    }
    @Override
    public void postStop() throws Exception {
        super.postStop();
        System.out.println("ChatRoom Stopped...");
    }
    @Override
    public void onReceive(Object msg) throws Exception {
    if (msg instanceof Register) {
        ActorRef userActor=getSender();
        //保存为聊天室用户列表
        clientActors.add(userActor);
```

仅供非商业用途或交流学习使用



```
    }else if (msg instanceof Message) {
        System.out.println("聊天室列表: "+clientActors.size());
        Message chatMsg = (Message) msg;
        if (chatMsg.isText()) {
            TextMessage textMsg = chatMsg.asTextMessage();
            //给每个User发送消息
            clientActors.forEach(client->client.tell(new SendToClient(textMsg),getS
                elf()));
    } else {
    BinaryMessage binaryMsg = chatMsg.asBinaryMessage();
    String filepath = "C:\\jeework\\HttpServer\\doc\\"+
System.currentTimeMillis() + ".jpg";
    CompletionStage<IOResult> comp = binaryMsg.getStreamedData().
runWith(FileIO.toPath(Paths.get(filepath)), mat);
    comp.thenAccept(r -> {
        BinaryMessage binMsg=
BinaryMessage.create(FileIO.fromPath(Paths.get(filepath)));
        //给每个User发送消息
        clientActors.forEach(client->client.tell(new
SendToClient(binMsg),getSelf()));
    });
    }
    }else if(msg instanceof Notice){
    Notice ter=(Notice)msg;
    ActorRef userActor=ter.getUserActor();
    clientActors.remove(userActor);
    getContext().stop(userActor);
    }}}
class User extends UntypedActor {
    private ActorRef clientActor;
    private ActorRef chatRoom;
    public User(ActorRef chatRoom) {
    this.chatRoom=chatRoom;
    }
    @Override
    public void postStop() throws Exception {
    super.postStop();
    System.out.println("UserActor Stopped...");
    }
    @Override
    public void onReceive(Object msg) throws Exception {
    if(msg instanceof Conn){
        this.clientActor=getSender();
        chatRoom.tell(new Register(),getSelf());
    }else if(msg instanceof SendToClient){
        SendToClient sendToClient=(SendToClient)msg;
        clientActor.tell(sendToClient.getMessage(),getSelf());
    }}}
```

User在收到Conn消息后，会将sender设置为对外通信的clientActor，然后给ChatRoom

仅供非商业用途或交流学习使用



发送注册（Register）消息，此时 ChatRoom 会将其保存在用户列表中。当 ChatRoom 收到消息后，它会首先判断消息的类型，假如是文本类型，那么其会将信息直接群发给所有 User，然后由 User 发送给对应的用户（通过 clientActor）。假如是二进制消息（比如图片），则其会将信息使用 Stream 读取出来并保存在本地，然后给所有 User 群发。

在服务端的代码里面，涉及一些自定义的消息指令，比如 Notice、SendToClient、Register、Conn，这四个指令主要用于约定一些 Actor 之间的行为，属于标识性的类型，本身不涉及业务逻辑，所以这里也不再给出定义代码。

### 2. 客户端实现

HTML5 为 WebSocket 提供了非常友好的 API，即使对前端技术不是太了解的工程师，也能快速开发出基于 WebSocket 的即时通信程序。而随着 HTML5 的快速发展，目前大部分浏览器（包括移动端）都支持了 WebSocket API，所有也基本上不用担心兼容性的问题。

HTML5 WebSocket API 的使用主要分为以下几个步骤：

1）**创建一个 WebSocket 连接对象**，在创建该对象时，需要指定服务端的 URL，它的格式类似于 HTTP，但是需要将协议换成 ws 或者 wss，比如 ws://127.0.0.1:8080/ws_demo。

2）**监听服务端消息**，当连接成功后，可以通过连接对象的 onmessage 回调函数接收服务端发送过来的消息，这个消息既可以是普通文本消息，也可以是二进制消息（Blob）。

3）**向服务端发送消息**，只要连接没有关闭，都可以调用连接对象的 send 方法向服务端发送消息，同样，这个消息既可以是普通文本消息，也可以是二进制消息（比如上传文件）。

4）**监听连接关闭**，当服务端关闭，或者出现异常时，连接都会被关闭，为了让客户端可以对此作出响应，可以调用 onclose 方法进行监听。

下面是使用 HTML5 WebSocket API 实现即时聊天的完整代码：

```
<!DOCTYPE html>
<html>
<head>
<title>欢迎来到首页</title>
<meta charset="UTF-8">
<script type="text/javascript">
    var client=new WebSocket("ws://127.0.0.1:8080/ws_demo");
    // 监听消息
    client.onmessage = function(ey) {
    var msgContent=document.getElementById("msgContent");
    var data=ey.data;
    if(data instanceof Blob){
    //二进制，这里是图片
    var fr=new FileReader();
    fr.readAsDataURL(data);
    fr.onload = function(evt) {
    var imgsrc = evt.target.result;
    msgContent.innerHTML+=("<img width='100px' width='140px' src='"+imgsrc+
        "'/><br/>");
```

仅供非商业用途或交流学习使用



```
        }
        }else{
        //文本
        msgContent.innerHTML+=(ey.data+"<br/>");
        }};
        client.onclose=function(evt){
        alert("连接关闭:"+evt);
        }
        function sendMsg(){
        var msgInput=document.getElementById("msgInput");
        var msgValue=msgInput.value;
        client.send(msgValue);
        msgInput.value="";
        }
        function uploadMyImg(){
        var myimg=document.getElementById("myimg");
        var _file=myimg.files[0];
        var fr = new FileReader();
        fr.readAsArrayBuffer(_file);
        fr.onload = function(evt) {
        var binaryString = evt.target.result;
        client.send(binaryString);
        }}
    </script>
    </head>
    <body>
        <div id="msgContent" style="border:1px solid;width:300px;min-height:400px">
        </div>
        <input type="file" id="myimg" name="myimg" onchange="uploadMyImg()" /><br/>
        <input type="text" id="msgInput"/><br/>
        <input type="button" onclick="sendMsg()" value="发送消息"/>
    </body>
    </html>
```

由于本书并不是前端方向的专业书籍，所以这里仅仅简单介绍一下这个页面的基本逻辑，并且默认大家都有 JavaScript 的基础。

当用户访问这个页面后，它会自动去请求 ws://127.0.0.1:8080/ws_demo，并且监听服务端发过来的消息，正常情况下，大家会马上看到聊天面板里显示出“您已进入聊天室 ...”这条消息。由于消息分为普通文本和二进制两种类型，所以在接收时需要做判断，假如是文本消息就直接展示，假如是二进制（Blob）消息，则通过 FileReader 读取内容后再展示。在这个程序中，我们的二进制消息就是图片。发送消息的过程也很类似，假如是文本消息，就直接调用 send 方法进行发送；假如是图片，则通过 FileReader 读取内容后再调用 send 方法进行发送。最后，当服务端出现异常，或者连接被关闭后，会触发 onclose 函数，此时我们可以给用户做出相应的提示。

现在，当运行服务端后，我们就可以通过 http://localhost:8080/ 访问到这个页面了，图 9-7 展示了两个“用户”的聊天效果。

仅供非商业用途或交流学习使用



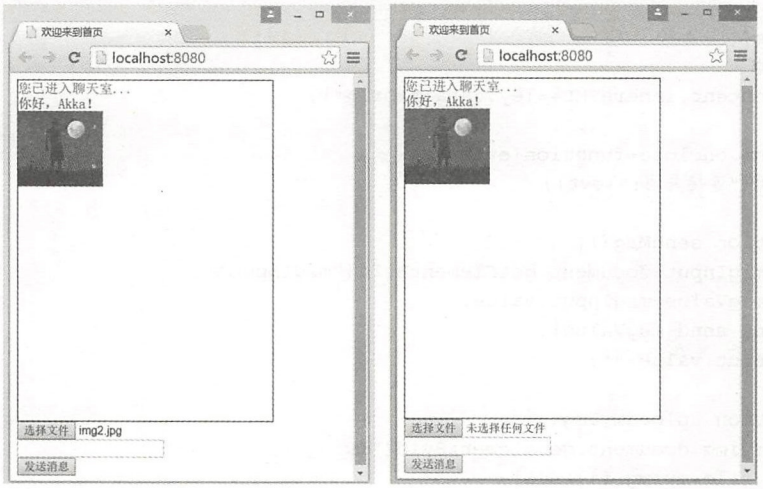



图 9-7 聊天室界面效果

### 3. 超时配置

在连接聊天室后，假如在一定的时间内没有发送或接收消息，那么它会自动关闭连接（ChatRoom 此时会收到 Notice 信息），这个时间称为最大活跃时间，它可以通过 akka.http.server.idle-timeout 来配置，默认为 60 秒。我们可以改得更大一点，比如 100 秒：

```
akka.http.server.idle-timeout=100 s
```

该配置对所有 HTTP 连接和 WebSocket 连接都起作用。

## 9.4 本章小结

Akka HTTP 服务构建于 Actor 和 Stream 的基础之上，提供了从客户端到服务端的全栈支持，诚然，它并不能算得上一个完备 Web 框架，但是这也给了我们进行自由定制化的机会。在本章中，笔者主要介绍了 Akka HTTP 的概念和基础用法，并借助 WebSocket 实现了一个简易的能收发文本和图片消息的即时聊天系统。

在下一章，我们来了解一个基于 Akka 的微服务框架——Lagom。

仅供非商业用途或交流学习使用

第 10 章 Chapter 10

# 微服务全家桶之 Lagom

最近几年，微服务的概念相当火爆，由于它确实能解决单体应用所带来的各种问题，所以越来越多的系统被架构师们“微服务化”了，或者正在“微服务化”的路上（笑）。经过几年的沉淀，大家对微服务开发积累了相当多的经验，一些共性问题也基本上都有了成熟的解决方案。同时，以 SpringCloud 为首的微服务框架迅速进入人们的视野，大家发现，使用这类“全家桶式”的框架进行开发是如此的省心。

而在本章中，笔者将为大家介绍另外一款“全家桶”式的微服务框架——Lagom，也让大家在做技术选型时有新的选择。

## 10.1 Lagom 简介

Lagom 框架是基于 Akka 和 Play 框架而构建的，不仅保留了 Akka 分布式、异步的特点，也提供了全套的开发、部署及运行等整套解决方案。

下面是 Lagom 几个比较明显的特点：

- **服务化**，提供构建地址透明的服务化接口的简单方式，这些接口支持同步和异步两种调用方式。
- **持久化**，提供基于 EventSource 的持久化功能，持久实体被分布在 Akka Cluster 的各个节点上，拥有良好的扩展性。
- **异步化**，Lagom 默认是异步的，这依赖于 Akka Stream 和 JDK8 的 CompletionStage API。
- **容器化**，提供一个开箱即用的容器环境，支持部署、监控和扩展。

仅供非商业用途或交流学习使用



**组件**

由于微服务本身的涉及面比较广，所以 Lagom 作为一个微服务平台，也自然依赖大量来自内部或者外部的开源组件。

- Akka 框架：如前面所说，Lagom 是基于 Akka 的，很多特性直接来自 Akka 自有实现，比如异步（Akka Stream）、持久化、发布订阅、分布式 & 集群（Akka Cluster）、基于消息驱动的响应等。
- Play 框架：Play 是一款 Web 框架，底层依赖 Akka-Actor 实现，是 Lagom 微服务的实现者。
- Cassandra：Lagom 使用 Cassandra 作为持久层。为方便调试，Lagom 开发环境内置了 Cassandra Server。
- Kafka：Lagom 使用 Kafka 作为消息中间件，以便让不同服务共享信息。为方便调试，Lagom 开发环境内置了 Kafka Server。
- Guice：Lagom 使用 Guice 框架做依赖注入。

## 10.2 Lagom 初探

前面讲过，Lagom 作为“全家桶”式的微服务框架而存在，所涉及的面也是比较广的，为了让大家先对它有个宏观的认识，我们不妨先跑一个 Demo 出来，看看它大致上是怎样运行的。

Lagom 支持 Maven 和 SBT 来进行搭建，这两个工具各有自己的优势，而对于 Java 程序员，可能 Maven 的使用率更高一点，所以后面我们会用 Maven 来搭建 Lagom 工程。

首先，输入以下命令生成官方 Demo：

```
mvn archetype:generate -DarchetypeGroupId=com.lightbend.lagom -DarchetypeArtifact
    Id=maven-archetype-lagom-java -DarchetypeVersion=1.3.4
```

图 10-1 所示是使用该命令时产生的部分重要信息。

```
D:\lagomwork>mvn archetype:generate -DarchetypeGroupId=com.lightbend.lagom -DarchetypeArtifactId=maven-archetype-lagom-java -DarchetypeVersi
on=1.3.4
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.2:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.2:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from [com.lightbend.lagom:maven-archetype-lagom-java:1.4.0-M1] found in catalog remote
Define value for property 'groupId': : com.demo.lagom
Define value for property 'artifactId': : lagomdemo
Define value for property 'version':  1.0-SNAPSHOT: : 1.0
Define value for property 'package':  com.demo.lagom: :
[INFO] Using property: serviceiClassName = Hello
[INFO] Using property: service1Name = hello
[INFO] Using property: service2ClassName = Stream
[INFO] Using property: service2Name = stream
Confirm properties configuration:
```

图 10-1　使用 Maven 命令搭建 Lagom 工程

仅供非商业用途或交流学习使用



如图 10-1 所示，在生成项目时，需要输入 groupId、artifactId、version 等信息，输入 Y 后，会生成一个如图 10-2 所示结构的 demo 工程。

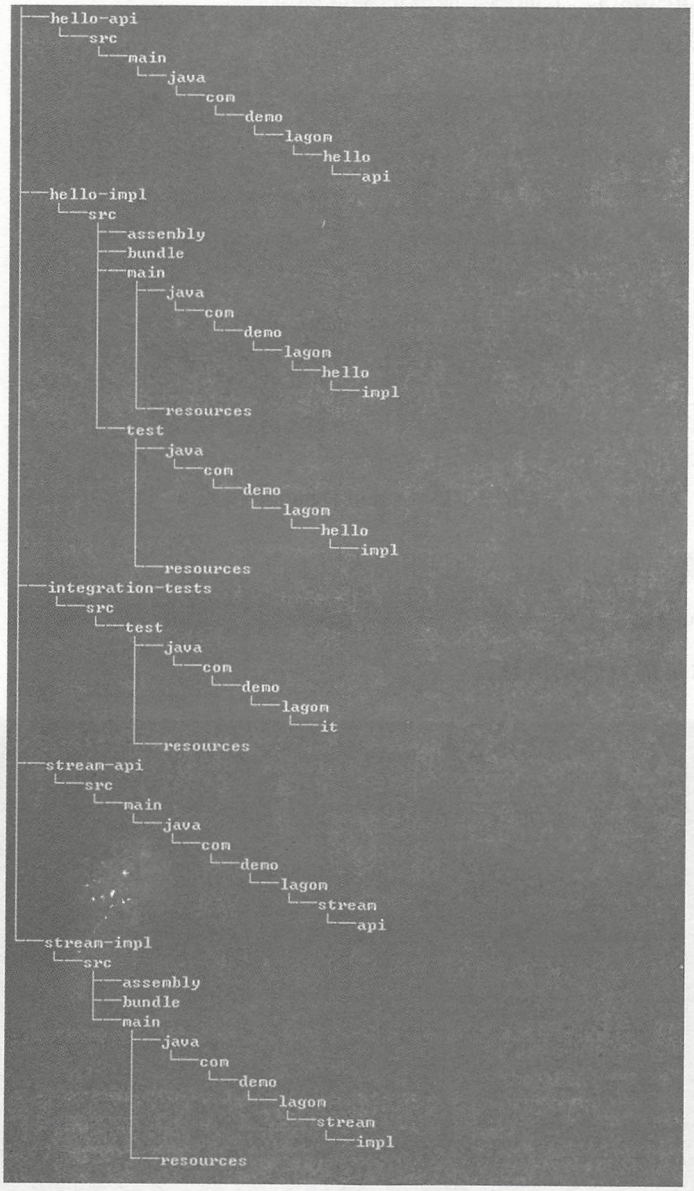



图 10-2　Lagom 版“Hello World”工程目录

如图 10-2 所示，该 demo 包含 5 个子项目，它们的功能如下：

- hello-api，一个“Hello World”式的服务接口；
- hello-impl，“Hello World”服务接口的实现；

仅供非商业用途或交流学习使用



- stream-api，一个 stream 的异步服务接口；
- stream-impl stream，异步服务接口实现；
- integration-tests，上面两个服务的集成测试。

它们的依赖关系如图 10-3 所示。

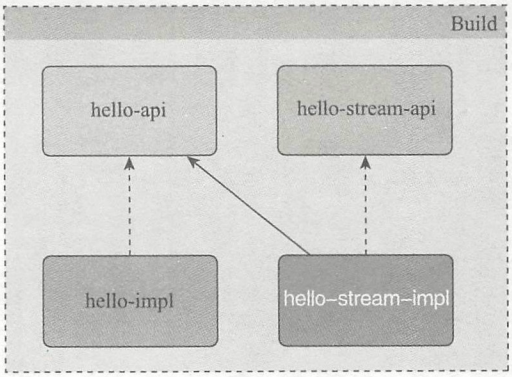



图 10-3　Lagom Demo 依赖关系

为了启动服务，我们可以进入项目根输入以下命令：

```
mvn lagom:runAll
```

图 10-4 所示是部分启动信息。

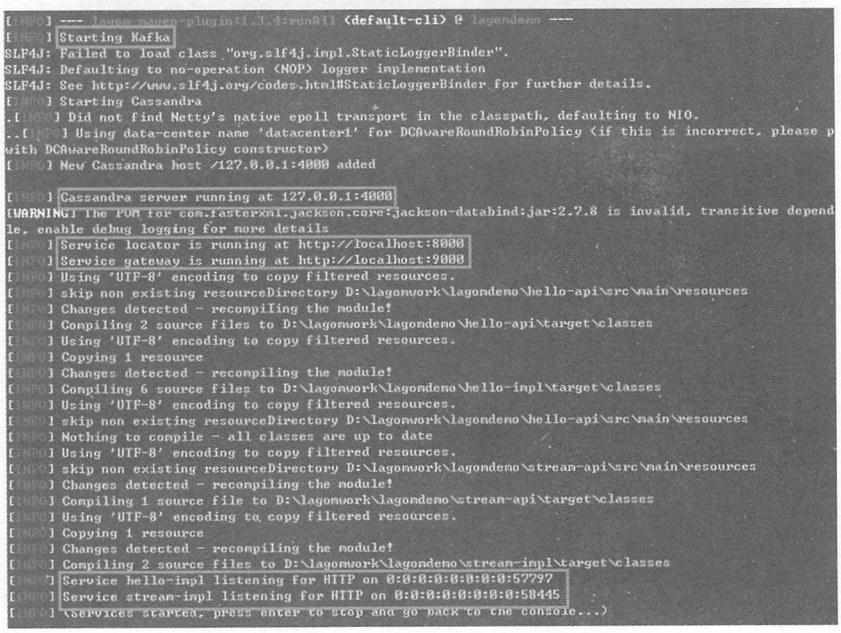



图 10-4　Lagom 项目启动信息

仅供非商业用途或交流学习使用



从图 10-4 所示可以看出，Lagom 项目启动时，大致会经历以下几个阶段：

1）启动 Kafka，作为消息中间件，默认端口是 9092；

2）启动 Cassandra Server，默认端口 4000；

3）启动 Service Locator（服务定位），默认端口是 8000；

4）启动 Service Gateway（服务网关），默认端口是 9000；

5）启动所有对外的业务服务。

此时访问如下链接，可以看到“Hello,World”。

```
http://localhost:9000/api/hello/World
```

这个 Demo 包含了 Lagom 大部分的使用场景，可谓“麻雀虽小 五脏俱全”，是学习的好素材。不过为了让大家能清晰地了解整个项目的结构和运行过程，接下来，笔者将会从头开始，一步步搭建起一个基础服务，并对每步涉及的知识点进行讲解。

## 10.3　搭建 Lagom 服务

本示例服务的场景比较简单，主要是实现“商品”的维护功能，比如查询、新增、修改等。通过这个例子，大家可以了解怎样定义和消费一个 Lagom 服务。

### 10.3.1　项目结构

首先，我们使用 Maven 搭建一套父子结构的工程，如图 10-5 所示。

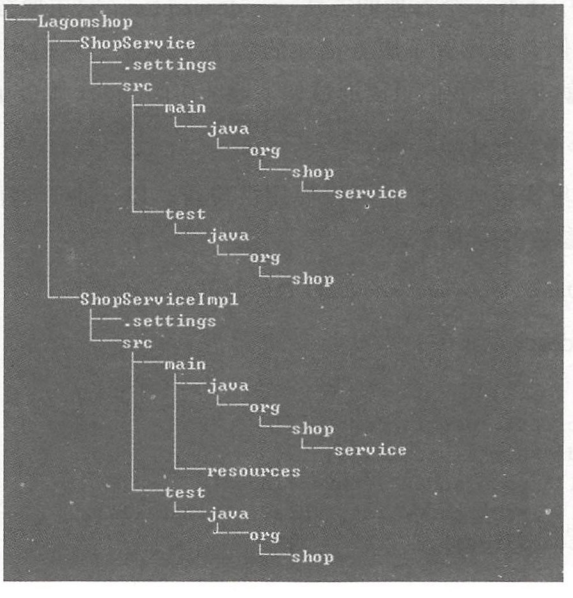



图 10-5　Lagom 示例项目结构

仅供非商业用途或交流学习使用



Lagomshop 是父项目，ShopService 和 ShopServiceImpl 分别是服务接口和服务实现的子项目，它们的 Maven 配置可直接参考官方 Demo，要注意的是：Maven 版本最好在 3.2.1 以上。

### 10.3.2 编写 Lagom 服务

#### 1. 定义服务接口

Lagom 服务会以接口的方式提供出去，在这个接口中，我们需要定义服务的内容和方式。下面我们看一个简单的示例：

```
public interface ItemService extends Service {

    ServiceCall<NotUsed, String> index();

    @Override
    default Descriptor descriptor() {
        return named("shop").withCalls(pathCall("/shop/index", this::index))
            .withAutoAcl(true);
    }
}
```

ItemService 接口会被定义在 ShopService 项目中，它必须继承 com.lightbend.lagom.javadsl.api.Service 接口，以表明自己是一个 Lagom 服务。ItemService 接口提供了一个 index 方法，用于获取网站首页信息（虚构），它和普通方法最大的区别是会返回 ServiceCall 对象，该对象用于描述服务调用的实体，其中的两个泛型分别表示请求和响应。

另外，这个接口中还有一个有趣的方法——descriptor，它主要描述了这个接口以什么方式去调用，这里我们首先设置了服务描述的名字（shop），然后用 pathCall 配置了 index 方法的访问路径，也就是说，我们可以通过如下地址访问 index 服务了：

```
http://localhost:9000/shop/index
```

假如需要在路径里面传入参数，可以使用占位符。以一个查询商品详情的接口为例，我们首先定义一个带参数的查询方法：

```
ServiceCall<NotUsed, String> showDetails(String id);
```

然后将其配置在 descriptor 中：

```
@Override
    default Descriptor descriptor() {
    return named("shop").withCalls(
        pathCall("/shop/index", this::index),
        pathCall("/shop/show/:id", this::showDetails))
        .withAutoAcl(true);
    }
```

现在就可以通过如下地址访问 showDetails 了：

```
http://localhost:9000/shop/show/123
```

假如需要传入多个路径参数，可以将 Call 配置成 /shop/:shopid/books/:gid，其中 :shopid 和 :gid 所对应的值会分别作为参数传入接口方法。大家可以看到，这种 URL 已基本符合 REST 风格，不过为了更贴近 REST 语义，Lagom 提供了 restCall 来进行配置，示例代码如下：

```
restCall(Method.GET,"/shop/index", this::index)
restCall(Method.GET,"/shop/show/:id", this::showDetails)
```

在上面示例中，我们都会把接口返回的 ServiceCall 的泛型设置为 <NotUsed, String>，即没有请求实体消息，此时默认是 Get 的请求方式（即使是非 restCall 配置的），假如把 NotUsed 设置为其他类型，就会变成 Post 请求方式。

### 2. 实现服务

接下来，我们需要在 ShopServiceImpl 中实现上述服务。

首先，定义一个类，实现 ItemService 接口：

```
public class ItemServiceImpl implements ItemService{
    @Override
    public ServiceCall<NotUsed, String> index() {
        return msg -> CompletableFuture.completedFuture("Hello, 欢迎进入购物商城!");
    }
    @Override
    public ServiceCall<NotUsed, String> showDetails(String id) {
        return msg -> CompletableFuture.completedFuture("Hello, 您查询的商品ID是:"+id);
    }
}
```

ServiceCall 的泛型规定了请求和响应的类型，在这里，请求和响应的类型分别是 NotUsed 和 String，这两种类型分别对应着 ServiceCall 函数的入参和出参：

```
CompletionStage<Response> invoke(Request request);
```

CompletionStage 接口表示一个异步计算的阶段或步骤，它最常见的实现类是 CompletableFuture，该类提供了多种异步执行的方法。在我们的示例中，仅仅返回一个已知的字符串信息，假如需要在其中加入一些其他异步处理，可以参考一下 CompletableFuture 更多的异步 API。

### 3. 绑定服务

在实现服务接口后，我们需要将其绑定在 Lagom 上，绑定方式如下：

```
public class ShopModule extends AbstractModule implements ServiceGuiceSupport{
    @Override
    protected void configure() {
        bindService(ItemService.class, ItemServiceImpl.class);
    }
}
```

然后将 ShopModule 配置在 application.conf 中：

```
play.modules.enabled += org.shop.ShopModule
```

### 4. 测试服务

现在，我们就可以输入以下命令启动服务了：

```
mvn lagom:runAll
```

图 10-6 所示是访问该服务的效果。

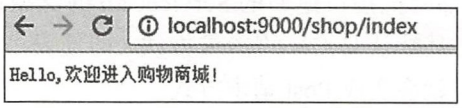



图 10-6　浏览器访问首页服务

前面提到过，假如将 ServiceCall 的请求泛型设置为其他类型，那么请求方式就是 post。下面改造一下 index 方法，在实现类代码中加上请求信息的打印，具体如下：

```
@Override
    public ServiceCall<String, String> index() {
    return msg -> CompletableFuture.completedFuture("Hello, 欢迎进入个人主页！ 请求信
        息:"+msg);
    }
```

那么此时再通过浏览器直接访问，则会出现如图 10-7 所示的界面。

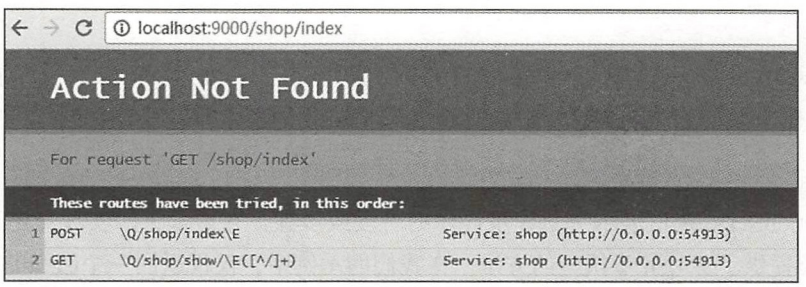



图 10-7　访问 Lagom 服务错误提示

大家会看到，/shop/index 已经成为 post 请求，为了方便观察，我们可以借助 Postman 来对它做请求，如图 10-8 所示。

如图 10-8 所示，我们将请求方式调整成 post，并且在 Body 选项卡中输入请求信息，然后点击 Send，就会在下面的区域看到响应信息。

假如请求或响应信息类型为其他自定义实体，那么 Lagom 会默认使用 Jackson 框架来做序列化或反序列化操作，比如说我们有个“添加商品”的接口实现如下：

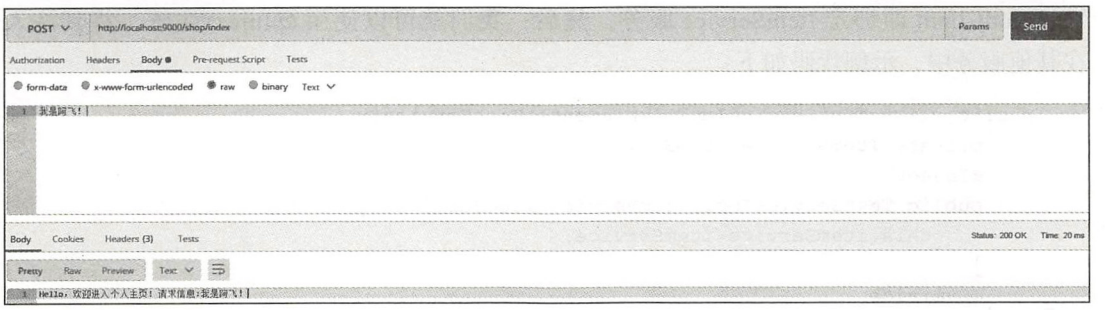



图 10-8　使用 Postman 测试 REST 服务

```
public ServiceCall<Item, Item> addItems() {
    return item -> CompletableFuture.completedFuture(item);
}
```

此时请求和响应信息都是 Item 类型，该类型拥有 id、name、price 等基础属性，那么在请求时，只需要构造标准的 JSON 格式就好了，如图 10-9 所示。

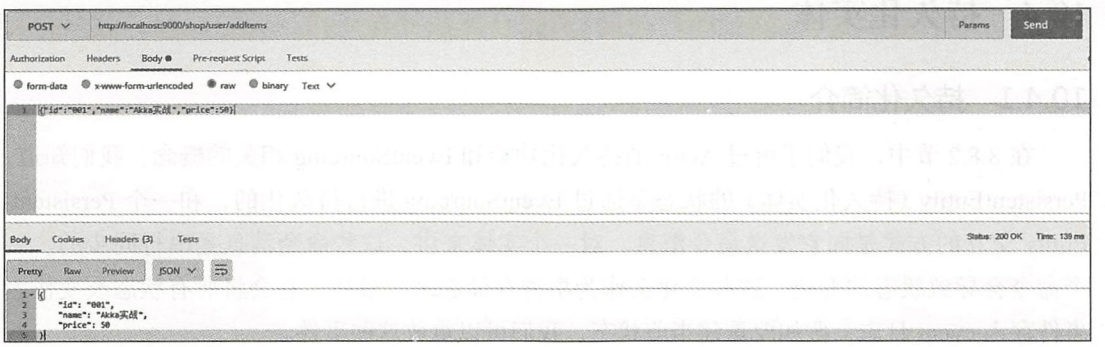



图 10-9　请求和响应数据的序列化和反序列化

### 5. 消费服务

在定义和实现服务后，我们就可以消费它了。消费 Lagom 服务分两步：绑定服务和注入服务。下面我们新开一个工程来说明这个过程。

首先，新建工程，让其依赖于 ShopService，并定义一个简单的服务接口（和实现），将其配置起来，具体如下：

```
public class TestModule extends AbstractModule implements ServiceGuiceSupport{

    @Override
    protected void configure() {
        bindService(TestService.class, TestServiceImpl.class);
        bindClient(ItemService.class);
    }
}
```

bindClient 即绑定 ItemService 服务，然后，我们就可以使用 @Inject 注解，将其注入在其他服务中，示例代码如下：

```
public class TestServiceImpl implements TestService{
    private ItemService itemService;
    @Inject
    public TestServiceImpl(ItemService itemService) {
        this.itemService=itemService;
    }
    @Override
    public ServiceCall<String, String> test() {
        return msg->itemService.index().invoke(msg);
    }
}
```

这样，我们就可以消费 ItemService 服务了，该示例的具体测试过程和结果这里不再赘述。

## 10.4 持久化实体

### 10.4.1 持久化简介

在 8.8.2 节中，我们了解过 Actor 的持久化功能和 EventSourcing 相关的概念，我们知道，PersistentEntity（持久化实体）的状态是通过 EventSourcing 进行持久化的，和一个 PersistentEntity 交互的方式是向它发送命令消息。对一个实体来讲，这些命令消息是串行同步的。一个命令会导致状态的变化，这些变化会作为事件存储起来，最终，它会将所有状态改变作为事件存入 event 日志，假如要重建当前状态，我们可以重放这些事件。

PersistentEntity 会自动的分布在服务的集群节点中，每个实体只在其中一个节点运行，向它发送消息时不需要知道它的位置，如果一个节点被停止，下一次发送消息时实体会在另外一个节点启动。假如新的节点被加入到集群，已存在的实体会重新负载到新的节点。

在 Lagom 中，我们可以选用以下数据库进行持久化：

- Cassandra
- PostgreSQL
- MySQL
- Oracle
- H2

其中，Lagom 开发环境已经内置了 Cassandra，不需要我们安装，但是在使用时不要忘了配置如下 maven 依赖：

```
<dependency>
    <groupId>com.lightbend.lagom</groupId>
    <artifactId>lagom-javadsl-persistence-cassandra_2.11</artifactId>
</dependency>
```

## 10.4.2　PersistentEntity API

要定义一个实体，必须继承 PersistentEntity 类，并实现 initialBehavior 方法，在该方法中，我们需要定义处理命令、事件和状态的函数，它大致上看起来像如下这样：

```
import com.lightbend.lagom.javadsl.persistence.PersistentEntity;
public class MyEntity
    extends PersistentEntity<Command, Event,State> {
    @Override
    public Behavior initialBehavior(Optional<State> snapshotState) {
    BehaviorBuilder b = newBehaviorBuilder(
    snapshotState.orElse(new State()));
    //...
    return b.build();
    }
}
```

PersistentEntity 需要三个泛型参数：其中 Command 表示命令，Event 表示事件，State 表示状态，它们都是能够被序列化的普通 Java 对象。

initialBehavior 是一个抽象方法，它返回实体 Behavior。Behavior 由当前状态、命令处理函数、持久化事件组成。

其中，处理函数包括两类：

- CommandHandler：命令处理，即当 PersistentEntity 收到命令信息时，会触发该函数，在这个函数中，我们需要调用 CommandContext.thenPersist 方法持久化事件，并根据业务需要响应信息。另外，当我们只做查询时，可以用 ReadOnlyCommandHandler 来进行处理。
- EventHandler：事件处理，即当事件持久化成功后，会触发该函数，此时应该进行状态的更新。

## 10.4.3　持久化示例

下面，我们仍然以 Lagomshop 为例来演示 PersistentEntity 的用法。

现在，我们添加两个接口——新增（或更新）商品和查询商品状态（价格），代码如下：

```
ServiceCall<Item,String> saveItem();
ServiceCall<NotUsed, Float> showItemState(String id);
```

然后定义命令、事件、状态，分别取名为 ItemCommand、ItemEvent、ItemState，它们的基本结构如图 10-10 所示（代码结构比较简单，不再给出）。

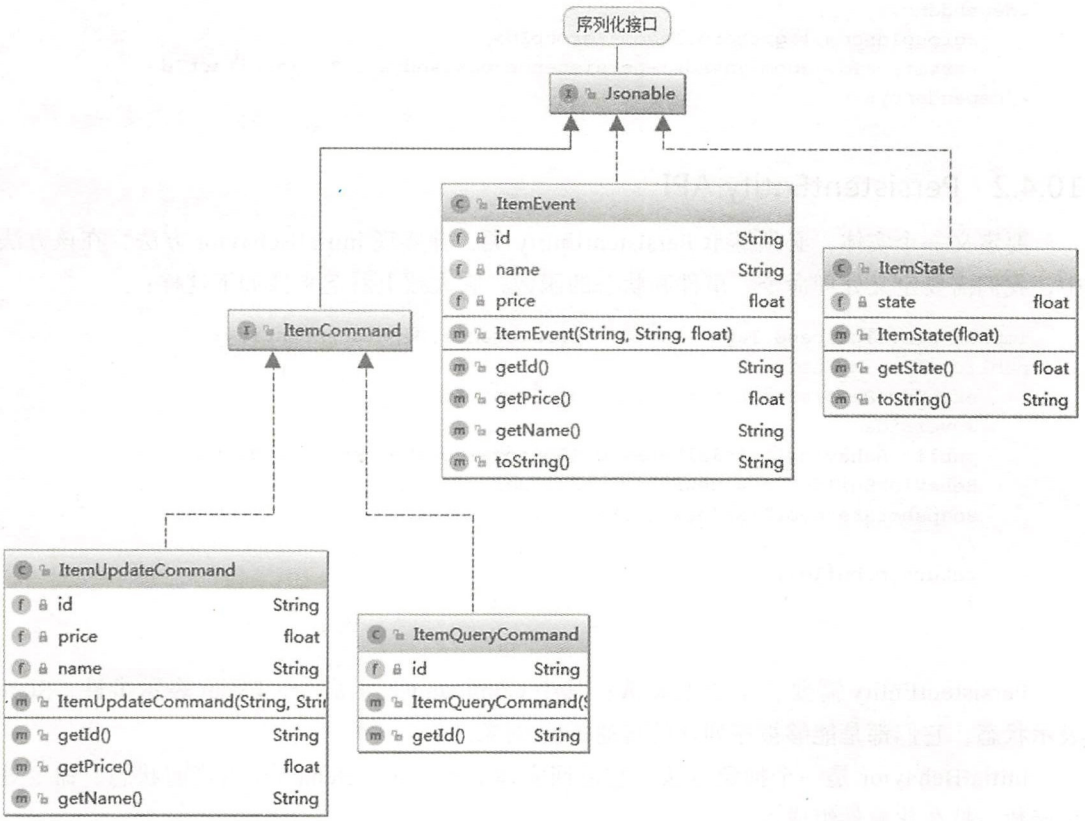



图 10-10　命令、事件、状态类结构

ItemCommand 下面有两个实现——ItemUpdateCommand、ItemQueryCommand，分别表示更新和查询，前者使用 setCommandHandler 函数进行处理，需要做持久化；后者使用 setReadOnlyCommandHandler 进行处理，不需要做持久化。

下面看看持久化实体的实现细节：

```
public class ItemEntity extends
    PersistentEntity<ItemCommand, ItemEvent, ItemState> {

    @Override
    public PersistentEntity<ItemCommand, ItemEvent, ItemState>.Behavior initial
        Behavior(
    Optional<ItemState> snapshotState) {
    BehaviorBuilder behaviorBuilder = newBehaviorBuilder(snapshotState
    .orElse(new ItemState(0)));

    behaviorBuilder.setCommandHandler(ItemUpdateCommand.class,(cmd,ctx)->{
    //通过entityId()可以得到实体的ID
    ItemEvent event=new ItemEvent(this.entityId(),cmd.getName(),cmd.getPrice());
```

```
        System.out.println("命令:"+cmd);
        return ctx.thenPersist(event,evt->ctx.reply("更新成功,"+event));
        });

        behaviorBuilder.setEventHandler(ItemEvent.class,evt->{
        ItemState currentState=this.state();
        ItemState state=new ItemState(currentState.getState()+evt.getPrice());
        System.out.println("事件:"+evt);
        return state;
        });

        behaviorBuilder.setReadOnlyCommandHandler(ItemQueryCommand.class,(cmd,ctx)->{
        ItemState currentState=this.state();
        System.out.println("状态:"+currentState);
        ctx.reply(currentState.getState());
        });
        return behaviorBuilder.build();
        }
    }
```

在这段代码中，我们首先初始化了商品的状态（state=0），然后使用下面几个方法处理命令和事件：

- setCommandHandler：当实体在接收到 ItemUpdateCommand 命令后，会调用该方法中注册的函数。在这个函数中，我们新建了一个事件，并调用 thenPersist 将其持久化，最后返回“更新成功”的提示。
- setEventHandler：当 ItemEvent 被持久化成功后，会调用该方法中注册的函数。在这个函数里中，我们先调用实体对象的 state() 方法得到当前状态，然后组装一个新的状态并将其返回，它将会被设置为本实体的最新状态。
- setReadOnlyCommandHandler：当实体在接收到 ItemQueryCommand 命令后，会调用该方法中注册的函数。该函数仅仅查出当前最新状态并返回。

大家会发现，我们在处理 ItemUpdateCommand 和 ItemQueryCommand 时，分别返回了 String 和 Float 类型的数据，这个实际上是在定义命令时通过实现 PersistentEntity.ReplyType 接口来指定的（每个命令都需要指定返回类型），代码如下：

```
public class ItemUpdateCommand implements ItemCommand , PersistentEntity.ReplyType
    <String>{...}
public class ItemQueryCommand implements ItemCommand ,CompressedJsonable, Persistent
    Entity.ReplyType<Float>{...}
```

接下来，我们希望能在服务层操作实体对象，这主要通过注入 PersistentEntityRegistry 对象来实现。

首先打开 ItemServiceImpl 类，改造一下构造函数，让 PersistentEntityRegistry 通过构造函数注入进去：

仅供非商业用途或交流学习使用



```
private final PersistentEntityRegistry persistentEntityRegistry;
    @Inject
    public ItemServiceImpl(PersistentEntityRegistry persistentEntityRegistry){
        this.persistentEntityRegistry = persistentEntityRegistry;
        persistentEntityRegistry.register(ItemEntity.class);
    }
```

然后调用 register 方法将实体类型注册起来，以便后续能操作它。对于查询状态的服务，我们要做的第一步就是通过 id 获取实体引用（PersistentEntityRef），然后调用 ask 方法和实体对象进行交互。根据前面的定义，此时应该向其发送 ItemQueryCommand 命令：

```
@Override
    public ServiceCall<NotUsed, Float> showItemState(String id) {
    return request->{
        PersistentEntityRef<ItemCommand> ref =
        persistentEntityRegistry.refFor(ItemEntity.class, id);
        return ref.ask(new ItemQueryCommand(id));
    };}
```

ask 方法会返回一个 CompletionStage 对象，其携带了实体返回的信息（ctx.reply）。

而对于更新操作，我们需要向其发送 ItemUpdateCommand 命令：

```
    @Override
    public ServiceCall<Item, String> saveItem() {
    return request->{
    PersistentEntityRef<ItemCommand> ref =
persistentEntityRegistry.refFor(ItemEntity.class,request.getId());
    return ref.ask(new ItemUpdateCommand(request.getId(),request.getName(),request.
        getPrice()));
    };}
```

启动服务后，我们使用 Postman 测试一下这两个功能。

首先，我们发送一个查询请求，id 为 123，由于此时并没有该实体，所以价格为 0，如图 10-11 所示。

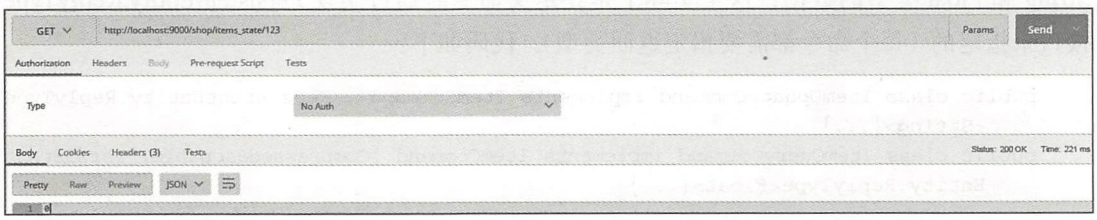



图 10-11　查询某个实体状态

然后，发送一个 post 请求，更新实体状态，该接口返回“更新成功”的提示，如图 10-12 所示。

仅供非商业用途或交流学习使用



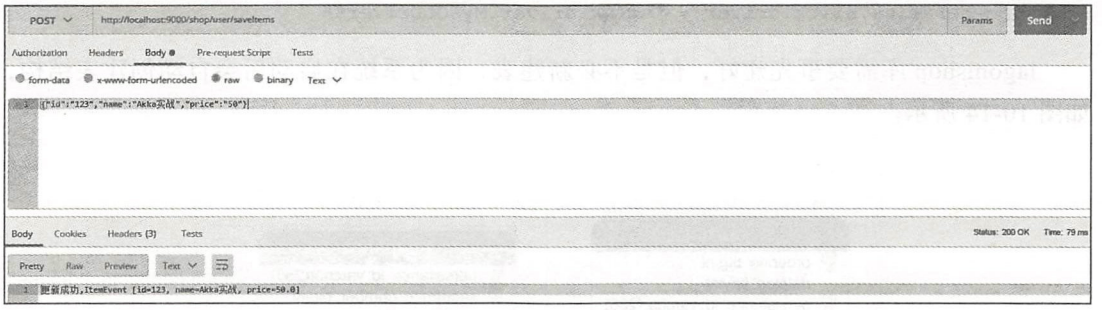



图 10-12　更新某个实体状态

为了验证刚才的更新是否真正的成功了，可以再次发送查询请求，查看返回的状态，如图 10-13 所示。

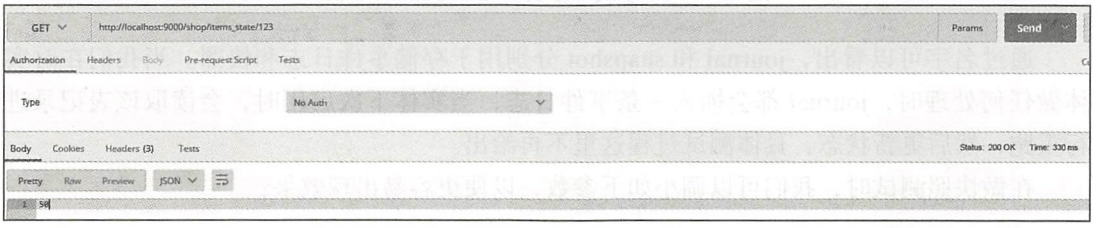



图 10-13　更新之后，重新查询实体状态

OK，测试通过。

## 10.4.4　MySQL 支持

除了 Cassandra 外，Lagom 持久化组件还支持其他关系型数据库，比如 MySQL、Postgre SQL、Oracle 等。下面我们以 MySQL 为例，看看它将以怎样的方式存储事件和快照。

首先，我们将 maven 配置中的 lagom-javadsl-persistence-cassandra 换成 lagom-javadsl-persistence-jdbc，并加上 MySQL 驱动包（略）：

```
<dependency>
    <groupId>com.lightbend.lagom</groupId>
    <artifactId>lagom-javadsl-persistence-jdbc_2.11</artifactId>
</dependency>
```

然后在 application.conf 中配置数据库连接信息：

```
db.default {
    driver = "com.mysql.jdbc.Driver"
    url = "jdbc:mysql://localhost:3306/lagomshop?characterEncoding=utf8"
    username="root"
    password="root"
}
```

仅供非商业用途或交流学习使用



```
jdbc-defaults.slick.driver = "slick.driver.MySQLDriver$"
```

lagomshop 库需要事先建好，但是不必新建表，因为系统在启动后会自动创建表结构，如图 10-14 所示。

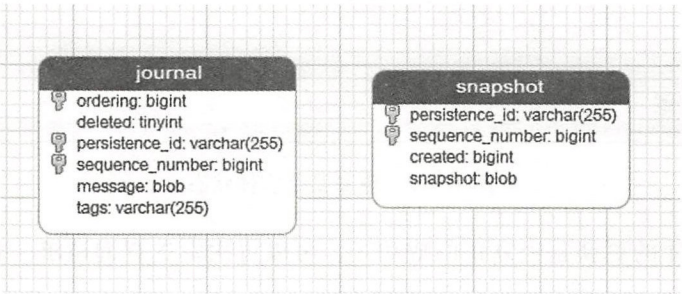



图 10-14　MySQL 事件日志和快照表结构

通过名字可以看出，journal 和 snapshot 分别用于存储事件日志和快照。当我们在对实体做任何处理时，journal 都会插入一条事件日志，当实体下次启用时，会读取该表记录进行重跑，然后更新状态，具体测试过程这里不再给出。

在做快照测试时，我们可以调小如下参数，以便更容易出现效果：

```
lagom.persistence.snapshot-after=5
```

该配置指定了，当持久化 5 个事件后，自动保存一次快照，此时大家会看到 snapshot 表里新增一条记录。当实体下次启用时，会首先通过快照来恢复状态，然后根据剩下的事件恢复到最新状态，缩短了整个恢复的时间。

## 10.5　Lagom 部署（SandBox）

Lagom 并不限定特别的生产环境，不过官方推荐使用 Lightbend ConductR 来部署、运行和监控 Lagom 服务。

Lightbend ConductR 可以理解为一个容器，里面包含了 Lagom 服务所需的运行及运维（包括监控）环境，它支持集群部署，同时提供了弹性扩展的能力。Lightbend ConductR 在商业上是收费的，不过在 Lightbend 官网上注册后，可以在 3 个节点内免费使用。ConductR 提供了一个“Sandbox”的预生产环境，方便我们在本地集群中调试服务，接下来，我们看看怎样在 Conduct Sandbox 部署 Lagom 服务。

### 10.5.1　安装 ConductR SandBox

为了能够使用 3 个免费节点，我们可以先去官网注册一个账号，该账号密码会在后续安装中用到：

仅供非商业用途或交流学习使用



```
https://www.lightbend.com/account/register
```

Lightbend ConductR 推荐的环境是 Ubuntu 16.04 LTS 或者 RHEL/CentOS 7，笔者的系统是 CentOS 7.3 64 位。在正式安装 ConductR SandBox 之前，需要先装好 JDK 环境，笔者的版本是 1.8。

接着，输入以下命令，下载并安装 Sandbox：

```
curl \
-u abc__demo@126.com -p -L -o /tmp/enterprise-suite.command \
"https://www.lightbend.com/product/conductr/createBuildFileAuth/linux"&& \
chmod u+x /tmp/enterprise-suite.command && \
/tmp/enterprise-suite.command
```

其中 abc_demo@126.com 是你注册的账号。输入命令后回车，会看到图 10-15 所示情形。

```
[root@slave1 ~]# curl \
>  -u a              com -p -L -o /tmp/enterprise-suite.command \
>  "https://www.lightbend.com/product/conductr/createBuildFileAuth/linux" && \
>  chmod u+x /tmp/enterprise-suite.command && \
>  /tmp/enterprise-suite.command
Enter host password for user 'a              com':
```

图 10-15　下载 ConductR Sandbox

输入密码后回车，会自动进行下载并安装，如图 10-16 所示。

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2104  100  2104    0     0    371      0  0:00:05  0:00:05 --:--:--   465

Adding Lightbend credentials to /root/.lightbend/...

Downloading the developer sandbox...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:--  0:00:01 --:--:--     0
  0     0    0     0    0     0      0      0 --:--:--  0:00:03 --:--:--     0
100 48.0M  100 48.0M    0     0  6078k      0  0:00:08  0:00:08 --:--:-- 12.5M

Extracting it...
Archive:  /tmp/conductr-cli-1.2.21-Linux-amd64.zip
  inflating: /usr/local/bin/conduct
  inflating: /usr/local/bin/sandbox
  inflating: /usr/local/bin/shazar
  inflating: /usr/local/bin/bndl

You have now installed the developer sandbox. Now execute:

  sandbox run 2.1.8 -n 3 -f visualization
```

图 10-16　安装 ConductR Sandbox

安装完成后，提示我们输入如下命令启动 sandbox:

```
sandbox run 2.1.8 -n 3 -f visualization
```

下面是部分重要的输出信息，如图 10-17 所示。

仅供非商业用途或交流学习使用





```
[root@slave1 ~]# sandbox run 2.1.8 -n 3 -f visualization
|-----------------------------------------------|
| Starting ConductR                             |
|-----------------------------------------------|
Network address aliases are required so that the sandbox can operate as a c
To add the network aliases sudo privileges are required.
sudo ifconfig lo:0 192.168.10.1 netmask 255.255.255.255 up
sudo ifconfig lo:1 192.168.10.2 netmask 255.255.255.255 up
sudo ifconfig lo:2 192.168.10.3 netmask 255.255.255.255 up
Bintray credentials loaded from /root/.lightbend/commercial.credentials
Retrieving https://dl.bintray.com/lightbend/commercial-releases/conductr-2.
[##################################################] 100%
Retrieving https://dl.bintray.com/lightbend/commercial-releases/conductr-ag
[##################################################] 100%
Extracting ConductR core to /root/.conductr/images/core
Extracting ConductR agent to /root/.conductr/images/agent
Starting ConductR core instance on 192.168.10.1..
Waiting for ConductR to start..
Starting ConductR agent instance on 192.168.10.1..
Starting ConductR agent instance on 192.168.10.2..
Starting ConductR agent instance on 192.168.10.3..
|-----------------------------------------------|
| Starting continuous delivery feature          |
|-----------------------------------------------|
Deploying bundle continuous-delivery..
Retrieving bundle..
Resolving bundle using [bintray_resolver, docker_resolver]
Loading bundle from cache typesafe/bundle/continuous-delivery
Bintray credentials loaded from /root/.lightbend/commercial.credentials
Resolving bundle typesafe/bundle/continuous-delivery
Bintray credentials loaded from /root/.lightbend/commercial.credentials
Retrieving https://dl.bintray.com/typesafe/bundle/typesafe/continuous-deliv
[##################################################] 100%
Loading bundle to ConductR..
[##################################################] 100%
Bundle 64affe9f9d126d31674f1408731b4e08 is installed
Bundle loaded.
Bundle run request sent.
Bundle 64affe9f9d126d31674f1408731b4e08 waiting to reach expected scale 1
Bundle 64affe9f9d126d31674f1408731b4e08 has scale 0, expected 1..
Bundle 64affe9f9d126d31674f1408731b4e08 expected scale 1 is met
```

图 10-17　Sandbox 启动信息

如图 10-17 所示，此时 ConductR 会在 3 个节点启动，然后加载所需要的 bundle，输入 conduct info，可以看到所有 bundles 的状态，如图 10-18 所示。

```
[root@slave1 ~]# conduct info
Licensed To: 974e6a08-5b28-4ffe-a139-bf763964646a
Max ConductR agents: 3
Expires In: 90 days (Tue 12 Dec 2017 16:48PM)
ConductR Version(s): 2.1.*, 2.1.0-beta.1
Grants: akka-sbr, cinnamon, conductr

ID       NAME                 TAG  #REP  #STR  #RUN  ROLES
64affe9  continuous-delivery  2.1.6     1     0     1  continuous-delivery
920f4ad  eslite               2.1.4     1     0     1  elasticsearch
cabaae7  visualizer           2.1.0     1     0     1  web
```

图 10-18　所有 bundles 信息

实际上，我们编写的 Lagom 服务也需要打包成 bundles 才能运行起来，具体打包过程会在后面进行讲解。

仅供非商业用途或交流学习使用





现在，我们可以访问以下地址，看看可视化界面：

```
http://192.168.10.1:9008/services/visualizer
```

具体效果如图 10-19 所示。

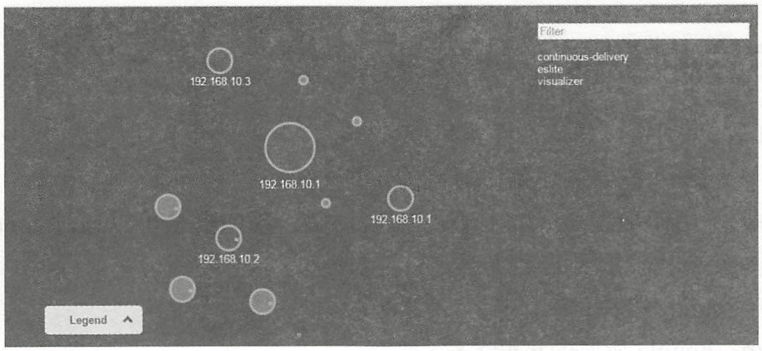



图 10-19　Lagom 服务可视化界面

假如要停止 sandbox，则执行 sandbox stop 命令。

## 10.5.2　部署 Maven 项目

要能让服务运行于 ConductR Sandbox，需要先将其打成 bundle 包，然后通过 conduct 命令载入，下面我们以 Maven 项目为例来演示这个过程。

打包的步骤大致如下：

1）创建 bundle 配置文件，一般命名为 bundle.conf。

2）创建启动脚本文件。

3）配置 Maven 打包插件。

首先，我们在 ShopServiceImpl 的 src 下创建 bundle 目录，然后新建 bundle.conf 文件，内容如下：

```
version = "1"
name = "shop"
compatibilityVersion = "1"
system = "shop"
systemVersion = "1"
nrOfCpus = 0.1
memory = 268435456
diskSpace = 200000000
roles = ["web"]
components = {
    shop = {
    description = "shop"
    file-system-type = "universal"
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用





```
start-command = ["bin/shop"]
endpoints = {
"shop" = {
bind-protocol = "http"
bind-port = 0
services = []
},
"akka-remote" = {
bind-protocol = "tcp"
bind-port = 0
services = []
}}}}
```

这个文件主要用于配置 bundle 信息，在启动后，这些信息可以通过 conduct info 命令查看到，下面是几个比较重要的属性：

- name：bundle 名称；
- system：bundle 所属的集群名称；
- nrOfCpus：运行该 bundle 所需的 CPU 的最小数量；
- memory：运行该 bundle 所需内存；
- diskSpace：bundle 扩展和配置所需磁盘空间；
- roles：运行该 bundle 的节点类型，默认为“web”；
- start-command：指向一个启动脚本文件；
- bind-port：服务实际绑定的端口，0 表示动态分配。

接下来在 bundle 目录新建 bin 目录，然后新建名为“shop”的启动脚本文件，内容如下：

```
#!/bin/sh
JVM_OPTS="-Xmx128m -Xms128m"
PLAY_SECRET=none
SHOP_BIND_IP=172.16.247.185
SHOP_BIND_PORT=9990
CONFIG="-Dhttp.address=$SHOP_BIND_IP -Dhttp.port=$SHOP_BIND_PORT -Dplay.crypto.
    secret=$PLAY_SECRET"
DIR=$(dirname $0)
java -cp "$DIR/../lib/*" $JVM_OPTS $CONFIG play.core.server.ProdServerStart
```

这个文件里面主要是配置启动参数和启动命令，其中 SHOP_BIND_IP 和 SHOP_BIND_PORT 主要配置了服务绑定的 IP 和端口。

然后在 src 下新建 assembly 目录，并新建 conductr-bundle.xml 文件，该文件主要用于打包配置，如打包格式、目录等，具体内容如下：

```
<assembly
xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用





```
http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3
http://maven.apache.org/xsd/assembly-1.1.3.xsd">
    <id>conductr-bundle</id>
    <formats>
    <format>zip</format>
    </formats>
    <baseDirectory>shop_v1</baseDirectory>
    <dependencySets>
    <dependencySet>
    <outputDirectory>shop/lib</outputDirectory>
    <outputFileNameMapping>${artifact.groupId}-${artifact.artifactId}-${artifact.ve
        rsion}${dashClassifier?}.${artifact.extension}</outputFileNameMapping>
    </dependencySet>
    </dependencySets>
    <fileSets>
    <fileSet>
    <outputDirectory></outputDirectory>
    <directory>src/bundle</directory>
    <excludes>
    <exclude>bin/**</exclude>
    </excludes>
    </fileSet>
    <fileSet>
    <outputDirectory>shop/bin</outputDirectory>
    <directory>src/bundle/bin</directory>
    <fileMode>0755</fileMode>
    </fileSet>
    </fileSets>
</assembly>
```

通过这个配置，我们将 src/bundle 下的文件原封不动的拷贝到 shop-v1 下，然后将所有依赖包，包括服务本身的 jar 包拷贝到 shop-v1/lib 下，目录结构如下：

```
├──shop-v1
|   ├──bin
|   └──lib
```

这个文件会被引用到 maven-assembly-plugin 插件配置中，所有插件配置如下：

```
<build>
    <plugins>
    <plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
    <descriptors>
    <descriptor>src/assembly/conductr-bundle.xml</descriptor>
    </descriptors>
    </configuration>
    <executions>
    <execution>
    <id>conductr-bundle</id>
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



```
        <phase>package</phase>
        <goals>
        <goal>single</goal>
        </goals>
        </execution>
        </executions>
        </plugin>
        <plugin>
        <groupId>com.lightbend.lagom</groupId>
        <artifactId>lagom-maven-plugin</artifactId>
        <configuration>
        <lagomService>true</lagomService>
        </configuration>
        <executions>
        <execution>
        <goals>
        <goal>renameConductRBundle</goal>
        </goals></execution></executions>
    </plugin></plugins></build>
```

然后输入 mvn package 打包，成功后会在 ShopServiceImpl\target 下看到一个名为 shop_v1-xxxx.zip 的包。

接着，我们在保证 Sandbox 启动的情况下，使用 conduct load 命令加载这个包，如图 10-20 所示。

```
[root@izbp159amgqk4mwaqjqh6vz test]# conduct load shop-v1-2875e91ec0b69bb38c5c82928f70f5149d7c138balefa6ac39fcfe1a6fd9d942.zip
Retrieving bundle..
Resolving bundle using [uri_resolver, bintray_resolver, docker_resolver]
Loading bundle to ConductR..
[##################################################] 100%
Bundle 2875e91ec0b69bb38c5c82928f70f514 is installed
Bundle loaded.
Start bundle with:        conduct run 2875e91
Unload bundle with:       conduct unload 2875e91
Print ConductR info with: conduct info
Print bundle info with:   conduct info 2875e91
```

图 10-20　加载“shop”包

加载成功后，使用 conduct info 命令，可以看到“shop”服务已经在列表中了，并且每个服务都有一个 ID，如图 10-21 所示。

```
[root@izbp159amgqk4mwaqjqh6vz test]# conduct info
Licensed To: 974e6a08-5b28-4ffe-a139-bf763964646a
Max ConductR agents: 3
Expires In: 80 days (Wed 06 Dec 2017 18:34PM)
ConductR Version(s): 2.1.*, 2.1.0-beta.1
Grants: akka-sbr, cinnamon, conductr

ID       NAME                 TAG  #REP  #STR  #RUN  ROLES
2875e91  shop                      1     0     0  web
cabaae7  visualizer           2.1.0     1     0     1  web
64affe9  continuous-delivery   2.1.6     1     0     1  continuous-delivery
920f4ad  eslite               2.1.4     1     0     1  elasticsearch
```

图 10-21　查看所有 bundle 信息

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用





如图 10-21 所示，shop 服务并没有启动，输入以下命令启动服务：

```
conduct run shop
```

这里的 shop 可以换成 Bundle ID（即 conduct run 2875e91 ）也是可以启动的。
另外，假如服务中依然用到了 cassandra，不要忘了加载并启动之：

```
conduct load cassandra
conduct run cassandra
```

现在，我们再用 Postman 对其进行测试，会发现服务已经 OK，如图 10-22 所示。

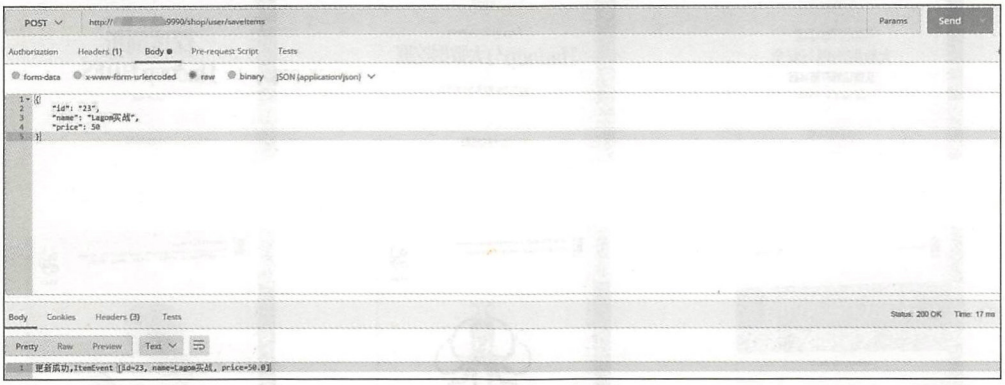



图 10-22　使用 Postman 测试“shop”服务

- 通过 conduct：命令可以管理 bundle 的整个生命周期、也可以很方便地查看日志、事件等信息，下面列出了常用的参数：
- version：打印版本号。
- info：打印所有 bundle 信息，假如加 Bundle ID 或名称参数，则打印具体 bundle 信息。
- load：加载 bundle 包。
- unload：卸载 bundle 包。
- run：启动 bundle（服务）。
- events：打印服务的事件信息。
- logs：打印服务的日志信息。

## 10.6　本章小结

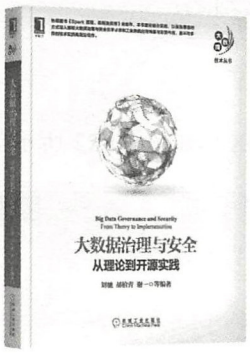
在本章中，首先介绍了 Lagom 的基本功能和特点；然后以一个“商品”服务为例，讲解了如何构建、实现和消费 Lagom 服务，并介绍了持久化实体的基本用法。在后面，演示了如何使用 ConductR Sandbox 安装和部署 Lagom 服务。

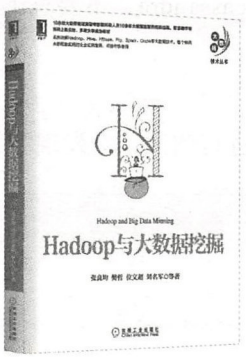


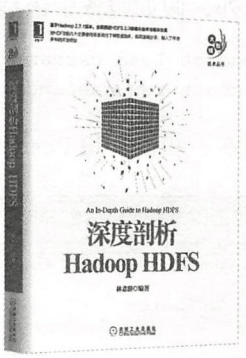
仅供非商业用途或交流学习使用

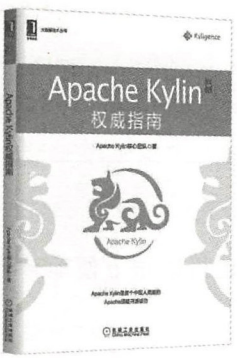
仅供非商业用途或交流学习使用

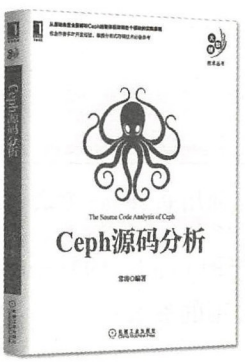
# 推荐阅读

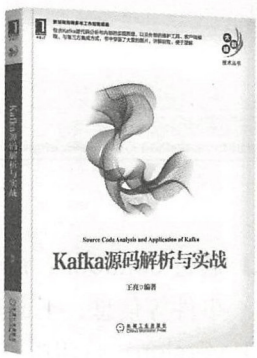


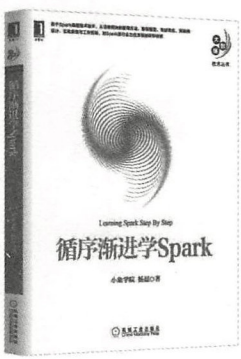



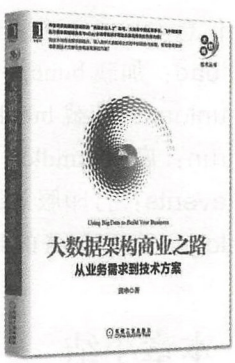










仅供非商业用途或交流学习使用

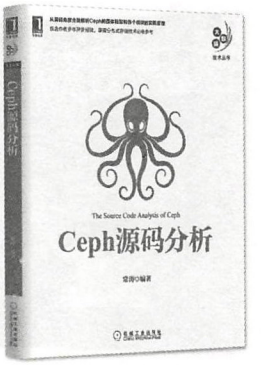
仅供非商业用途或交流学习使用

# 推荐阅读



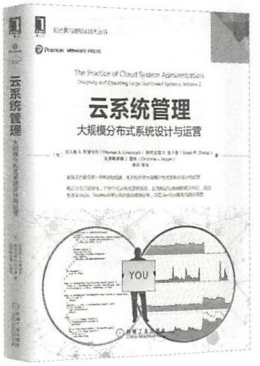


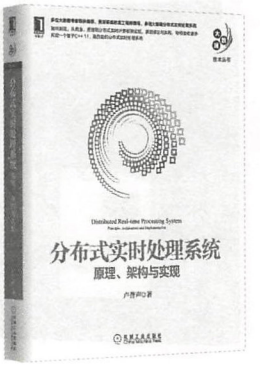








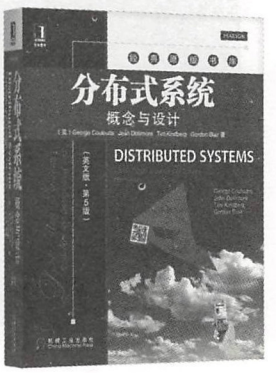




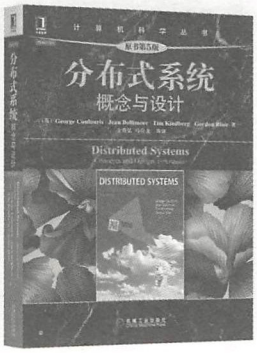




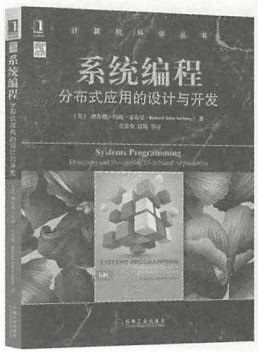



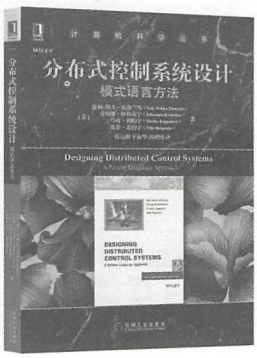

仅供非商业用途或交流学习使用

# 推荐阅读

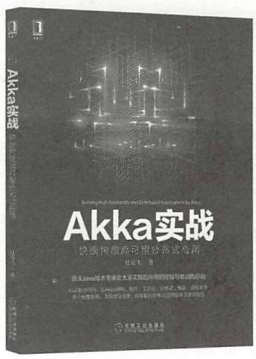

分布式系统
概念与设计
Distributed Systems
DISTRIBUTED SYSTEMS


深入分布式缓存
从原理到实践


系统编程
分布式应用的设计与开发
Systems Programming
SYSTEMS PROGRAMMING

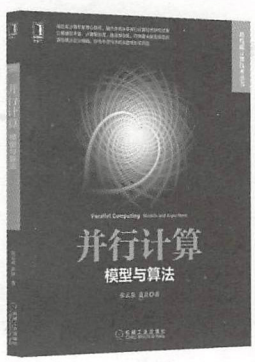

分布式控制系统设计
模式语言方法
Designing Distributed Control Systems
DESIGNING DISTRIBUTED CONTROL SYSTEMS


分布式实时处理系统
原理、架构与实现


Akka实战


高性能并行珠玑
多核和众核编程方法
High Performance Parallelism Pearls
High Performance Parallelism Pearls


并行计算的编程模型
Programming Models for Parallel Computing


并行计算
模型与算法



## 作者简介

### 杜云飞

资深 Java 技术专家，上海小虫科技合伙人兼 CTO，在 Akka 方面有丰富的实践经验。

从事软件开发和架构相关工作近 10 年，擅长 Java 体系下各项技术，在大型企业软件平台（如制造业 ERP、财税云平台、智慧城市服务平台、电商系统等）的开发和架构方面经验丰富。

曾穿梭于各个技术岗位之间，并乐此不彼，期间独立创业近两年。多次受邀为惠普、通用等企业做技术分享、培训和咨询。

封面设计 · 陈潇然

Building High Availability and Distributed Applications by Akka

Akka 是一款优秀的分布式并发框架，虽然它是基于 Scala 语言实现的，但我们却可轻松地将其运行在 JVM 上，在不改变现有架构的基础上支持更高的并发量。另一方面，Akka 是一款轻量级开源技术，它既具备良好的性能，也具备优秀的扩展性，非常适合应用在微服务架构中。写书是一件很有挑战的工作，尤其是 IT 技术类原创图书，相信云飞这本书能为读者提供帮助。

——黄勇 特赞科技 CTO/《架构探险》作者

学习并掌握一个分布式并行应用框架，已经成为当下对普通程序员的基本要求。Akka 以其高性能、高容错性，恰巧可以作为一个极佳的基础学习框架。作者在 Akka 方面不仅有丰富的实战经验，并且善于将实战中的各种要点和最佳实践，以通俗易懂的方式呈现在本书中。对于广大程序员而言，本书是对官方文档极佳的补充和解释，强烈推荐！

——陆舟 社区技术专家 /《Struts2 技术内幕》作者

Akka 的文档稀缺、学习曲线偏高、难以上手……现在，国内终于有了一本 Akka 的专著。本书系统讲解了 Akka 的方方面面，既有 Akka 本身的特性，也有 Akka 的周边生态，理论与实践并重，值得一读。

——周立 《Spring Cloud 与 Docker 微服务架构实战》作者

目前市面上关于 Akka 的资料比较少，为了帮助读者更好地了解并使用 Akka，作者从实战的角度全面介绍了它，读者朋友们可以借助本书快速构建自己的高可用的分布式并行应用，非常值得一读。

——占军（占小狼） 美团点评基础架构组技术专家

上架指导：计算机/程序设计
ISBN 978-7-111-59685-1
9 787111 596851
定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn